



# PaaS实现与运维管理

基于Mesos+Docker+ELK的实战指南

余何 编著

電子工業出版社  
Publishing House of Electronics Industry  
北京•BEIJING

## 内 容 简 介

本书主要阐述了云计算中 PaaS 平台的实现与运维管理，分为四大部分，分别是概念模型、基础资源、平台实现与运维管理，共十五章。第一部分阐述了运维与开发之间的关系、这层关系存在的矛盾，以及 PaaS 是如何有效缓解其矛盾的；第二部分勾勒出了数据中心的计算、网络、存储三大资源的主干，避免让人陷入上层的种种产品中；第三部分通过开源产品来构建一个完整的 PaaS 平台，包括资源管理、任务调度、计算单元打包、分布式协调、日志集中等；第四部分对运维管理进行了实践。

本书适合运维工程师、运维管理人员，以及希望在 PaaS 上运行分布式、可伸缩、高可用的后端开发工程师阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。  
版权所有，侵权必究。

## 图书在版编目（CIP）数据

PaaS 实现与运维管理：基于 Mesos + Docker + ELK 的实战指南 / 余何编著. —北京：电子工业出版社，2016.2

ISBN 978-7-121-27502-9

I. ①P… II. ①余… III. ①程序语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字（2015）第 262151 号

责任编辑：孙学瑛

印 刷：

装 订：

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×1092 1/16 印张：27.75 字数：620 千字

版 次：2016 年 2 月第 1 版

印 次：2016 年 2 月第 1 次印刷

印 数：3000 册 定价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 [zltz@phei.com.cn](mailto:zltz@phei.com.cn)，盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

服务热线：（010）88258888。

# 推荐语

---

本书最大的特点是理论联系实际，深入浅出地讲解了 PaaS 的实现方法，涉及当下非常热门的 Docker、Mesos 技术，更难能可贵的是，作者同时分享了珍贵的运维实践经验，我为读者能看到本书的精彩内容而感到高兴。

“KVM 虚拟化实践”公众号维护者、珠海金山西山居系统运维经理 肖力

每一名 IT 工程师都有自己的 PaaS 平台梦，这才有近年来风行的 GAE、CloudFoundry、Docker。然而 PaaS 要在企业中应用推广，绝不是改个容器或虚拟机那么简单。本书以 PaaS 为题，内容却覆盖了平台建设的理论基础、技术实现、配套系统和流程管理，是作者多年实践经验的精华所在，堪称大型企业应用运维平台化的指导用书。

微博运维团队系统架构师 饶琛琳

PaaS 越来越受到重视，它是未来真正的高效运维利器。本书作者有着丰富的一线运维经验，在本书中从 PaaS 平台的多个角度阐述了其核心原理与技术实现。值得称赞的是，作者还给出了应用系统如何适应 PaaS 及对 PaaS 应用的运维管理，让我们对 PaaS 平台有了系统而全面的认识。

优维科技创始人、互联网运维杂谈 王津银

国内有不少运维同行在自己的业务领域尝试引入私有 IaaS 技术，因此催生了不少分享 IaaS 体系理念及实践的资料和书籍。伴随着愈演愈烈的竞争，互联网业务对运营及运维能力的要求也会不断攀升，大家会慢慢意识到私有云的概念停留在 IaaS 层面是远远不够的。国内分享 PaaS 体系的理念及实践的资料和书籍并不多，本书不但对 PaaS 理论及基础概念介绍得比较清楚，而且以平安科技的实践经验为背景，对广大运维及运营开发同行会有很大的帮助。

腾讯游戏蓝鲸产品中心总监 咖啡党

云服务是当前炙手可热的话题，现在业务使用云服务是趋势，百度内部的业务线也很久就在推广 Matrix、PaaS 技术。本书内容通俗易懂，讲解由浅至深，读者既能从书

## PaaS 实现与运维管理：基于 Mesos + Docker + ELK 的实战指南

中了解到云服务的现状，又能从细节上了解 PaaS 的实现原理。作者本身对 PaaS 有着丰富的实战经验，结合书中丰富的源代码，能使读者快速踏入 PaaS 领域。

百度国际化首席架构师、广告变现技术团队负责人 谢朴锐

相较 IaaS 而言，PaaS 更为丰富、可操作性更强，作者精选多种流行、前沿的 PaaS 工具、产品及运维自动化工具，再辅为基础理论知识和个人多年运维行业技术经验写成本书。本书文字朴实易懂、图文并茂，适合运维同仁们细细阅读。

开放运维联盟联合主席、高效运维技术社区创始人 萧田国

# 推荐序一

---

第一次见作者是在 2007 年年中，他刚到平安科技入职。初次交谈，他给我最大的印象是虚心好学，积极上进。当时为了提升团队整体的计算机专业水平，我们进行了约一年的内部相互学习和培训，重点放在计算机最底层的计算机组成和程序运行原理上。记得第一次安排他做的培训题目是从软件破解的角度去了解计算机软件。他用了两周来学习和准备就可以大致讲明白编译运行类程序的内存布局和进行破解的方法。这种自学的激情和效率不是所有人都能做到的，正是这种好学上进的精神成就了今天的他。其实，要把计算机方面的工作做到极致，必须要有这种精神。

作者与我在一个 IT 团队共事五年，我们一起经历过系统运维一线的种种复杂而烦琐的工作：应用环境构建、问题应对、重大事故处理、运维管理……我们曾经通宵达旦地解决事故背后的疑难问题，曾经为解决运维中的资源管理、监控、自动化、工作协同等问题一起设计并开发运维工作平台。多年前，我们有一个想法：我们的团队亲身经历所获取的经验和教训，整理下来就是一本很好的书。一年前，作者提到自己准备写一本反映 IT 运维工作本源和真谛的书。当时觉得，IT 运维工作者确实需要这样一本书。市面上的运维书籍多在讲解如何去使用某个软件或者系统，却很少提及如何面对多种多样的系统、软件、硬件，更谈不上从本质和道理层面上去讲解运维的本真。对于任何一个事物，如果我们掌握了它的本质，则应对起来会更加游刃有余。好比程序员读懂了 *Programming from ground up*，OS 管理员读懂了《计算机的心智——操作系统之哲学原理》，创业者读懂了《创业维艰》。虽然，现在本书把重点放在了 PaaS，但多从计算机技术本质角度提出问题和解决问题，也算是回归了运维本真的思路。

IT 运维有两种复杂度：一是应用规模大，一个应用要应对海量访问，例如上了规模的互联网运维；二是应用数量多，碎片化小应用特别多，例如大企业中大量的异构小应用和复杂的网络拓扑。这两种复杂度带来的问题总是让人头疼。面对这些复杂的问题，如何选择合适的方法和技术进行应用的快速构建、资源配置、信息管理、监控、操作自动化等，并没有一致的答案。要有答案，需要你充分考虑和分析所在环境的团队素质、外部资源支持、应用特点等因素。作者以自己的亲身工作经验为背景，对这些重要内容进行了讲解。

引用爱恩斯坦的一句话：Any intelligent fool can make things bigger, more complex, and

## PaaS 实现与运维管理：基于 Mesos + Docker + ELK 的实战指南

more violent. It takes a touch of genius and a lot of courage to move in the opposite direction, 意思是自命不凡的学者总会把事物变得更大、更复杂和更剧烈，而在相反方向上努力是需要不凡的天赋和极大的勇气的。

IT 运维工作者，你懂的！

TechSure 创始人兼 CEO 温海波

2015 年 10 月

# 推荐序二

---

云计算正在中国普及，在 IaaS、PaaS、SaaS 的三层服务里：IaaS 的标准相对成熟；SaaS 百花齐放、新应用层出不穷；PaaS 作为衔接 IaaS 与 SaaS 的平台服务层，现在越来越受到人们的重视，专门提供 PaaS 的创业公司也越来越多了。

本书介绍了比较常见和重要的 PaaS 系统，包括基于容器的操作系统虚拟化技术 Docker、分布式协调系统 ZooKeeper、资源管理系统 Mesos、服务调度框架 Marathon、大数据处理架构 Spark、日志搜索分析系统 ELK 等。Docker、Mesos、Spark、ELK 等系统在美国都有专门的创业公司如 Docker、Mesosphere、Databricks、Elastic 等知名公司在开发并提供技术支持服务，这些公司受到了风险投资界的追捧。更有专门的公司开发和维护这些系统，这是对其生命力和前途的背书。这些系统已经在业界得到广泛应用，每一位对云计算感兴趣的技术人员都应该了解这些系统。

以日志管理为例，一家公司的服务器、网络设备、应用系统每时每刻都在产生日志，大公司的 IT 系统可能每秒产生超过 10 万条日志，每天产生的日志量达到 TB 级。这些日志包含了极其重要的运维信息和业务信息，但分散在服务器和网络设备上，需要登录每台设备上查看，存储空间满了就被覆盖、删除，管理非常不方便。一些大公司建立了日志管理系统，把分散在各台设备上的日志采集上来，集中管理，并提供查询、分析、统计等功能。因为日志是非结构化数据，传统的使用数据库的处理方式不适合日志处理，于是出现了 ELK 这种采用实时搜索引擎处理日志的开源系统。本书详细介绍了 ELK。

另外，本书也涵盖了数据中心运维技术和管理如配置、监控、变更等，对 GAE、Cloud Foundry、Heroku 等国外热门的 PaaS 也有介绍，对运维工程师大有裨益。

本书作者在平安科技从事运维管理工作多年，经历了平安科技从金融 IT 到互联网金融的转变。互联网强调快速迭代，金融 IT 强调稳定合规，能把这对矛盾体结合在一起，实属不易。作者在这方面积累了丰富的经验，现在把这些宝贵经验分享出来，是对 IT 运维界的贡献，希望本书能够对云平台运维工程师有所帮助。

日志易创始人兼 CEO 陈军

2015 年 9 月

# 前言

---

## 古往今来随风去，书剑相伴两袖空

笔者在运维领域耕耘十余载。2007 年加入平安集团旗下的科技公司，2011 年主导了业内最大的应用迁徙与架构变更，2012 年开展 IT 运维管理变革，打通横向条线，实现了技能融合。光阴荏苒，日月如梭，运维往事历历在目，我们流过汗，熬过夜，摊过事，也拿过奖，运维是一个从无到有、日积月累、不断提升的过程，也是一个需要耐得住寂寞，顶得住压力的行当，在此与正奋斗在运维一线的伙伴们共勉。

平安 IT 经历了企业信息管理部、国际领先 IT 服务公司、互联网金融三个历史阶段，每一次蜕变都紧随时代步伐，拥抱技术革新，创造业务价值。从企业 ERP、PC 互联、移动互联、云计算到大数据，运维人一直在奔袭，从未停歇。企业信息管理时期的运维人飞行于各大城市的上空，每一次发版、变更就是一次长途远行，分散的管理模式简单直接，但随着业务发展、规模扩大，企业对系统稳定性、运营成本控制的要求越来越高，随之而来的是 IT 集中管理时期，这个时期追求一流的国际 IT 服务管理水平，构建独立数据中心，实现开发架构、安全标准与运营规范的全面统一，该时期形成的 IT 管理实践堪称行业标杆。2013 年全面进入互联网金融时期，移动互联、云计算、大数据的火热发展，业务渠道、流量入口、用户终端的改变，促使我们又一次站在了历史的新起点。

互联网金融在运维方面上演了一场“双城记”。一边是充满贵族气质、为荣誉而战的“英伦君主立宪”，在这个领域内严格执行 IT 管理规范，稳若磐石，滴水不漏，保证绝对稳定与安全；而另一边是自由、民主喧哗下的“法国大革命”，为了满足互联网下业务系统的高并发、高吞吐、版本多变的需求，应用不再拘泥于一致框架、规范与准则，多样选择，以快为先。本书的核心内容就是在这样的背景下诞生的，我们无法“一刀切”地构建承载所有应用类型的基础环境，只有准确定义应用标准，以一种兼容并存的方式在应用架构上迭代地朝轻量级、微模块方向发展，通过技术、管理双层标准来调和“君主立宪”与“民主共和”之间的矛盾，转化成优势互补、经验传承、全面共赢，最终完成新时期的历史使命。这个过程是无法一步到位、一蹴而就的，而是以一种螺旋式形态探索前进的，除了在原有管理规范上引入新技术探究，还需要开发同步配合在应用架构上进行改造，基础架构将从环境运维逐渐转变成平台研发，以提供功能粒度细、服务自助化的运维平台来满足上层应用需求。



## 通明大道去浮华，千辛历尽回本真

PaaS 并不能解决 IT 运维的所有问题，它对应了“民主共和”的部分，提供了一个用户自助的开放形式平台来满足部分应用需求。PaaS 也不是从一开始就从解决异地容灾、流量切换等数据中心级别功能上设计的，而是在兼容当前企业应用架构、满足资源分配、计算单元打包、版本发布控制等基础功能后，再做进一步功能延伸的。PaaS 并不是一种新兴技术，而是对以往运维经验的总结，利用容器等技术对开发、运维工作流的重新编排。

IT 技术更新发展快，新词汇层出不穷，特别是在云计算、大数据、移动互联下衍生了大量的产品，让运维人应接不暇，刚刚掌握 Hadoop，又出 Spark，才标准化 ActiveMQ，又有 Kafka，OpenStack 的 IaaS 体系才得以了解，又有各种平台下的 PaaS……如果要熟练掌握每一门技艺，则将是一个疲于奔命的过程，而这个过程大部分都是表面的产品架构与配置，我们会发现一旦深入其中，所有技术却看似一致。对于运维人员来说，亘古不变的始终是底层的三大基础资源计算、存储、网络，这些知识包括：程序的链接、加载与运行；操作系统下的 CPU、内存、I/O 资源管理；基本的 TCP/IP 协议栈等。随着技术层级越多、复杂性越高，运维人员只有透过眼花缭乱的“新技术”看到其内在一致的本质，在底层长期沉淀，理解好程序是如何调用三大基础资源，找到运维技术的本真，才可以做到最终的不变应万变，这也印证了运维是一个要耐得住寂寞的行当。

最后让我们回到运维管理上，今年国内互联网上发生了几起影响比较严重的运维故障，一时间在圈内引起关于“再流弊的技术，也抵不过一次事故”等的科技杂谈，可见管理在运维上的重要性。管理是一门艺术，而这门艺术在运维上并不是形而上学、趋于神秘主义的，也不是绝对的“封”“杀”“控”，以怀疑主义、不信任方式来管控的。运维管理是人、流程与工具三方面的有机结合。“民主共和”型运维管理往往依赖于人的自觉性，而“君主立宪”则重在流程管控，朝两个极端偏倚而忽视它们与工具的结合都将不能从根本上解决问题。我们会发现越是技术实力强、自信的人越容易犯运维错误，而再完善的流程制度，如果烦琐到让人感觉“无意义”，那也就无从执行了。工具是人与流程之间的桥梁，但一定要注意这个工具不是管控的“枷锁”，而是在满足运维管理需求下关注一线运维人员用户体验的“帮手”。对于管理层，它是上层意志的体现，但对于一线运维人员，它是一个效率工具，除了包含精准配置信息、标准变更步骤，还要囊括人性的知识分享、社交互动等功能，它是为一线运维人员服务、以人为本的。

## 内容大纲

本书分为四大部分，分别是概念模型、基础资源、平台实现与运维管理，共十五章。各部分之间没有必然联系，读者可依据关注点和个人兴趣来阅读。对于需要系统理解运维及 PaaS 的读者，建议遵循本书的章节顺序阅读。

第一部分——概念模型：阐述了运维与开发之间的关系、这层关系存在的矛盾，以及 PaaS 是如何有效缓解其矛盾的。这部分介绍了公有 PaaS 平台的特征，以及其开放性与约束

性，列出 12-Factor 规范来说明应用系统应当遵循的规则，这样才能适应于在 PaaS 上运行。

第二部分——基础资源：勾勒出了数据中心的计算、网络、存储三大资源的主干，避免让人陷入上层的种种产品中。对于已熟悉数据中心三大资源的运维人员来说，本部分是资源的总体概述，让你重拾内在本质。若你是一名运维新兵，则请以此为纲要寻找外部资源来继续深入学习；开发人员可通过本部分了解到日常运维工作所管理对象的基本内容。

第三部分——平台实现：通过开源产品来构建一个完整的 PaaS 平台，包括资源管理、任务调度、计算单元打包、分布式协调、日志集中等。通过学习本部分的内容，读者可以实现一个可扩展、自定义的开放 PaaS 平台，这个扩展部分包括了各自企业内部的集成部署流程、应用灰度发布、平台门户管理等方方面面的内容。

第四部分——运维管理：对运维管理进行了实践。运维管理的核心是配置管理，一个好用、易用的配置管理系统将直接影响上层监控管理、变更管理及事件管理，决定一个企业运维品质的好坏。

### 感谢

首先特别感谢我的太太李嘉，在过去一年里，写作几乎占用了我所有的周末及其他休息时间，你承担起了家务，并对我与余多多悉心照顾，没有你的理解、宽容与支持，本书无法完成。感谢姐夫喻立新、姐姐何碧，你们在本书插图上给予了我很大的帮助。特别感谢饶琛琳、杨永帮利用周末进行审订稿件，感谢梁山在 12-Factor 上对我的帮助，感谢策划编辑孙学瑛老师对我的鼓励，感谢责任编辑虾米（张国霞）的校对、排版与指导。

感谢我的公司平安科技，给予了我一个更大的平台，让我驰骋在 IT 运维大草原上得以一览全貌。感谢我的上级胡玮、朱永忠、李毅对我工作的支持，感谢我的同事王欣、于泳、宋楹柯、蓝景全、江锐、常明、黄文建、唐文祥、陈顺星、彭俊清、陈春润对我的帮助。感谢事件处理组的小伙伴们：王耀武、莫广华、陈秋浩、郑司营、赵宝磊、吴磊、林国峰、张浙栋、夏永燕、孟佩佩、罗颖胜、倪沛榆、丁江，感谢基础架构篮球队。感谢所有在互联网上帮助过我的朋友们。

余何

2015 年 11 月

# 目 录

---

## 第一部分 概念模型

第 1 章 分布式 PaaS 平台介绍	2
1.1 什么是 PaaS	2
1.1.1 开发与运维之间的困局	2
1.1.2 DevOps 的自动化	3
1.1.3 云计算的 IaaS	4
1.1.4 PaaS 的到来	4
1.1.5 PaaS 的约束与开放	4
1.1.6 PaaS 解决的具体问题	5
1.2 什么是分布式计算	6
1.2.1 分布式计算与 PaaS	6
1.2.2 分布式平台的挑战	7
第 2 章 PaaS 模型与特征	10
2.1 主流 PaaS 平台架构	10
2.1.1 谷歌 GAE	10
2.1.2 AEB	11
2.1.3 Cloud Foundry	13
2.1.4 Heroku	14
2.2 PaaS 与 12-Factor	15
2.2.1 基准代码 (Codebase)	15
2.2.2 依赖 (Dependency)	16
2.2.3 配置 (Config)	17
2.2.4 后端服务 (Backing Services)	18

2.2.5	构建 (Build)、发布 (Release)、运行 (Run)	19
2.2.6	进程 (Process)	20
2.2.7	端口绑定 (Port Binding)	21
2.2.8	并发 (Concurrency)	21
2.2.9	快捷性 (Disposable)	22
2.2.10	开发/生产环境等价 (Dev/Prod Parity)	23
2.2.11	日志 (Log)	24
2.2.12	管理进程 (Admin Process)	25
2.3	PaaS 与 Reaction 宣言	26
2.3.1	响应 (Responsive)	26
2.3.2	韧性 (Resilient)	26
2.3.3	弹性 (Elastic)	27
2.3.4	消息驱动 (Message Driven)	28

## 第二部分 基础原理

第 3 章	计算资源	30
3.1	图灵机与冯·诺伊曼模型	30
3.2	服务器的种类	34
3.3	一切都是二进制	37
3.3.1	整数表示法	38
3.3.2	文本表示法	39
3.3.3	音频信息表示法	41
3.4	操作系统——计算机系统的指挥官	42
3.4.1	操作系统解决的问题	42
3.4.2	企业级操作系统	43
3.4.3	服务器虚拟化	47
3.5	进程——资源聚合的抽象体	49
3.5.1	计算单元的构建	49
3.5.2	计算请求的拆解	51

第4章 网络资源	53
4.1 协议分层	53
4.1.1 OSI 网络体系模型	54
4.1.2 OSI 与 TCP/IP 协议簇	55
4.1.3 交换、选路与传输	56
4.2 网络物理设备	58
4.2.1 连线与接口	59
4.2.2 二层交换机	62
4.2.3 路由及三层交换	63
4.2.4 四~七层网络设备	64
4.2.5 现实网络构成	65
4.3 网络逻辑拓扑	65
4.4 对网络拓扑的考虑	66
4.5 对物理布线的考虑	67
4.6 网络虚拟化与 SDN	70
第5章 存储资源	73
5.1 俯瞰存储系统	73
5.1.1 数据存储功能分类	73
5.1.2 文件存储的三个层级	74
5.2 磁盘与磁盘阵列	77
5.2.1 硬盘的物理构造	77
5.2.2 磁盘阵列	79
5.2.3 SCSI 协议	81
5.3 存储、计算分离	82
5.3.1 磁盘柜与盘阵	82
5.3.2 FC 存储网络	83
5.3.3 FC 协议栈	86
5.3.4 FC 寻址过程	87
5.3.5 FC 交换机与适配器	88
5.3.6 FCoE 与 iSCSI	89

5.4 存储访问类型.....	90
5.4.1 NAS 与 SAN.....	90
5.4.2 分布式存储.....	92

## 第三部分 平台实现

第 6 章 平台功能与架构.....	96
6.1 平台运维需求.....	96
6.1.1 软件配置.....	96
6.1.2 服务部署.....	97
6.1.3 服务发现.....	97
6.1.4 监控恢复.....	97
6.2 平台功能划分.....	97
6.3 平台高阶架构.....	100
6.4 企业应用迁移.....	102
6.4.1 企业应用很“厚重”.....	102
6.4.2 应用部署架构.....	102
6.4.3 企业应用调整.....	104
第 7 章 计算单元 Docker.....	108
7.1 Docker 介绍.....	108
7.1.1 Docker 是什么.....	108
7.1.2 Docker 术语.....	109
7.1.3 Docker 安装.....	111
7.2 Docker 容器命令.....	112
7.2.1 run 命令.....	112
7.2.2 start 命令.....	115
7.2.3 stop 命令.....	116
7.2.4 restart 命令.....	116
7.2.5 attach 命令.....	116

7.2.6	ps 命令	116
7.2.7	inspect 命令	117
7.3	Docker 镜像命令	119
7.3.1	search、pull、push 命令	120
7.3.2	commit 命令	120
7.3.3	image、diff、rmi 命令	121
7.3.4	save、load、export、import 命令	121
7.4	Docker 网络与链接	122
7.4.1	Docker 网络模式	122
7.4.2	pipework 管理网络	125
7.4.3	容器链接与数据卷	127
7.5	Dockerfile	129
7.5.1	基本指令集	130
7.5.2	环境指令集	131
7.5.3	数据指令集	132
7.5.4	ENTRYPOINT 指令	132
第 8 章	分布式协调 ZooKeeper	134
8.1	ZooKeeper 介绍	134
8.1.1	ZooKeeper 是什么	134
8.1.2	ZooKeeper 架构	135
8.1.3	数据模型	136
8.1.4	监听与通知	139
8.1.5	API 集合	139
8.1.6	会话	140
8.1.7	观察者	141
8.2	ZooKeeper 使用	141
8.2.1	快速安装	141
8.2.2	基本操作	143
8.2.3	配置参数	145
8.2.4	动态重配置	149

8.2.5	监控	152
8.3	ZooKeeper 进阶	157
8.3.1	分组与权重	158
8.3.2	Paxos 算法	159
8.3.3	ZAB 协议	163
8.3.4	分布式协调场景	165
第 9 章	资源管理 Mesos	167
9.1	Mesos 介绍	167
9.1.1	资源管理需求	167
9.1.2	Mesos 的起源	169
9.2	Mesos 架构与 workflow	169
9.2.1	Mesos 架构组件	169
9.2.2	Mesos 资源管理的工作流程	170
9.3	Mesos 安装配置	172
9.3.1	安装预先准备	172
9.3.2	构建 Mesos	173
9.3.3	启动 Mesos	174
9.3.4	高可用 Mesos	178
9.3.5	Slave 移除限速	182
9.4	Mesos 运维	183
9.4.1	认证管理	183
9.4.2	监控管理	186
9.4.3	容器网络限速	192
9.4.4	Framework API 限速	194
9.4.5	Restful 接口	195
9.4.6	配置参数	196
9.5	Mesos 资源分配	197
9.5.1	DRF 算法	197
9.5.2	DRF 权重	199



第 10 章 服务调度框架 Marathon	200
10.1 Marathon 介绍	200
10.1.1 服务调度平台	200
10.1.2 Marathon 实体模型	201
10.2 Marathon 使用	203
10.2.1 安装启动	203
10.2.2 运行 Shell 程序	204
10.2.3 运行远程资源	208
10.2.4 Artifact Store	209
10.3 Docker 容器运行	211
10.3.1 前提准备条件	211
10.3.2 端口资源分配	212
10.3.3 容器端口分配	215
10.3.4 其他使用方法	216
10.4 Marathon 管理	217
10.4.1 应用组	217
10.4.2 策略约束	219
10.4.3 健康检查	221
10.4.4 应用部署	223
10.4.5 事件总线	227
10.4.6 命令行参数	229
10.5 服务发现	231
10.5.1 服务发现方法	231
10.5.2 Marathon 方案	232
10.5.3 Mesos-DNS	235
10.5.4 Bamboo	239
10.6 Chronos 作业调度	241
10.6.1 作业调度框架	241
10.6.2 安装运行	241
10.6.3 作业示例	242
10.6.4 REST API	243

第 11 章	大数据调度框架 Spark	245
11.1	Apache Spark 介绍	245
11.1.1	Apache Spark 是什么	245
11.1.2	Lambda 架构	246
11.1.3	Spark 生态系统	247
11.2	Spark 数据处理	248
11.2.1	Spark 运行模式	248
11.2.2	Spark Standalone 模式	252
11.2.3	Spark on Mesos	255
11.2.4	Spark Streaming	257
第 12 章	日志集中管理 ELK	261
12.1	日志集中	261
12.1.1	日志集中介绍	261
12.1.2	日志集中架构	262
12.1.3	日志集中框架	264
12.2	Logstash	266
12.2.1	Logstash 介绍	266
12.2.2	快速安装	267
12.2.3	配置说明	269
12.2.4	部署架构	282
12.2.5	处理流程	285
12.2.6	input 插件	286
12.2.7	output 插件	292
12.2.8	filter 插件	296
12.2.9	codec 插件	299
12.3	Elasticsearch	300
12.3.1	基本概念	300
12.3.2	安装与使用	304
12.3.3	REST API	305
12.3.4	集群设置	309

12.3.5 备份恢复.....	314
12.3.6 监控管理.....	315
12.4 Kibana .....	317
12.4.1 Kibana 介绍 .....	317
12.4.2 discover 功能 .....	319
12.4.3 visualize 功能.....	324
12.4.4 Dashboard 功能.....	327

## 第四部分 运维管理

第 13 章 配置管理 .....	330
13.1 配置管理系统分析 .....	331
13.1.1 服务模型进行分层.....	331
13.1.2 各 IDC 团队发现 CI .....	332
13.1.3 IDC 管理团队定义 CI 属性 .....	333
13.1.4 确定 CI 之间的关联 .....	336
13.2 配置管理系统设计 .....	338
13.2.1 用户界面设计.....	339
13.2.2 权限控制、规则定义和 OPENAPI .....	341
13.2.3 数据模型的设计 .....	343
13.3 配置管理数据准确性的保证 .....	345
13.3.1 识别 CI 的 OWNER .....	345
13.3.2 识别 CI 的生命周期、关联运维流程 .....	346
13.3.3 数据有效性的审计 .....	346
第 14 章 监控管理 .....	348
14.1 运维监控管理的问题与价值 .....	348
14.1.1 监控管理的无形价值 .....	349
14.1.2 监控平台建立的基础 .....	350
14.1.3 监控管理的 WANT 原则 .....	350

14.2	对运维监控平台的需求分析	352
14.2.1	一次监控过程，调度、规则、告警	352
14.2.2	数据图形化：百分位裁剪、趋势分析、正态分布	358
14.2.3	开源的借鉴与选择：Zabbix 和 Nagios	361
14.2.4	商业与开源：最后的决策	372
14.3	JMX 监控原理解析	373
14.3.1	JMX 的体系结构	374
14.3.2	一个完整的 JMX 体系架构实例	376
14.3.3	通过 JMX 访问 WebLogic Server MBean	379
14.4	SNMP 监控原理解析	383
14.4.1	SNMP 协议解析：MIB 库与消息类型	383
14.4.2	使用 SNMP4J 实现服务器监控	386
14.4.3	Linux 下的监控实现：NET-SNMP	390
14.4.4	MIB 库浏览工具：ManageEngine	391
第 15 章	运维管理	392
15.1	服务级别管理，IT 与业务的一致性	392
15.1.1	客户满意度与期望	393
15.1.2	服务目录——IT 服务的菜谱	396
15.1.3	从宏观到可操作性的服务	397
15.2	变更管理，使服务有效传递	399
15.2.1	变更控制的角色、阶段	399
15.2.2	变更管理的六个原则	401
15.2.3	变更分类与风险定级	402
15.2.4	表单、步骤、模板与日历	405
15.3	事件管理	409
15.3.1	分类管理与评价体系	409
15.3.2	任务分发、协同与时效	411
15.3.3	内部上报要求	412
15.3.4	重大事件处理	413

15.4 人员管理：开放与分享.....415

    15.4.1 企业社交管理.....415

    15.4.2 目标管理，做好绩效.....417

    15.4.3 知识管理，人员成长.....417

    15.4.4 时间管理，个人效率.....420

15.5 PaaS 下的运维发展之路.....421





# 第一部分 概念模型

PaaS 提供的服务是直接面向开发人员的，是数据中心级的应用逻辑“大容器”，它解决了长久以来开发与运维之间存在的“矛盾”。第一部分让我们从开发、运维的关系起航，游历各大主流 PaaS 平台，并从“大容器”角度思考应用设计的基本原则，从而勾勒出清晰的 PaaS 概念模型。

## 分布式 PaaS 平台介绍

### 1.1 什么是PaaS

#### 1.1.1 开发与运维之间的困局

我们将视角投放到一个大中型 IT 企业中，这里有两个独立的团队对生产运行的服务负责，它们分别是开发团队和运维团队。开发团队负责服务的业务逻辑的正确执行，运维团队负责服务的稳定运行（暂且让我们抛开产品经理、项目经理与业务用户）。开发运维团队之间的配合要到什么程度才堪称优秀呢？理论上说这两个团队的交互越少、等待时间越短，两个团队的配合就越好。

在实际情况中我们却常常看到开发人员不得不陷入复杂的基础架构工作之中，在域名、IP、组件、防火墙、操作系统等基础配置工作中煎熬。开发人员关注的是应用运行环境的交付速度、质量，他们其实并不需要熟悉运维人员在基础层面做了些什么。而运维人员则少不了地抱怨开发人员需求太多、太急而无法按期交付，他们面向需求不一的多个开发团队，开发团队应用的网络区域、操作系统版本、中间件组件等存在不一致性，而运维团队的业务又涉及网络、计算、存储、中间件等多个领域，在管理上涉及监控、容量、变更等，这样，问题被进一步复杂化，两个团队之间的协作越来越困难。

开发人员的主要工作是编码并将它们放到测试、生产环境中运行，若关于应用发布的任意任务（发布版本、修改页面、资源扩容等）都因运维而被拖延，则会造成整个产品的发布延迟、运营成本增加，更可怕的是打乱了产品创新的节奏。在移动互联网时代，产品发布的效率直接影响其成功率。

在很长一段时间内运维人员都希望通过标准化、自动化、自助化三步走的方式解决以上问题，而现实总是残酷的，没有任何方法论指导他们。标准化的愿景在开发需求多样化的前提下湮灭；没有标准何以达到自动化，自动化停留在脚本级的小范围内，其对效率的影响无法立竿见影；自助化建立在自动化、标准化之上，而对基础组件的自助，反过来又会威胁到基础资源的安全性。



### 1.1.2 DevOps的自动化

开发、运维之间的“困局”很快引起了一阵 DevOps 风，大家都厌恶了在基础资源上的等待，厌倦了重复、枯燥的人工运维，为了按时交付软件产品和服务，开发人员和运营人员必须紧密合作，他们希望能够形成一套方法论，通过可编程方式来管理和控制基础架构资源，轻松、愉快地解决问题。

DevOps 定义了如下明确的目标：

- 更小、更频繁的变更意味着更少的风险；
- 让开发人员更多地控制生产环境；
- 更多地以应用程序为中心来理解基础设施；
- 定义简洁明了的流程，尽可能地自动化；
- 促成开发人员与运营人员的协作。

部分开发人员转到 DevOps 的实现工作中，特别是全面自动化运维工具的实现工作。Chef、Puppet、Saltstack 等 DevOps 工具陆续出现，它们有一致的中心控制-代理的应用架构，提供了一套可移植、去差异、管理成千上万台服务器的方法。但在可移植性上，必须重新定义一门抽象的语言来覆盖原各个 OS 平台。应用升级、文件替换、版本部署等运维操作都可以通过这门语言组合到一个模板中，从而实现复用、快速交付。开发人员被告知有越来越多的工具帮助我们快速交付，但运维工具的新语言本身又带来了复杂性。谁对这组语言模板负责，到底是开发人员还是运维人员？如果是运维人员来负责，那么仅仅做到了自动化，由于受标准化程度及运维工具本身的复杂度影响，在不同环境下的效率和收益会截然不同。而如果是开发人员负责，那么他们还需要花很长的时间去掌握一门运维“语言”来实现自动化，在模板、脚本执行过程中发生的任何问题，最终还是要辗转转到运维人员处，问题貌似又回到了原点。

DevOps 期望通过一套方法论与工具来填补开发工作与运维工作之间的沟壑，如图 1-1 所示。

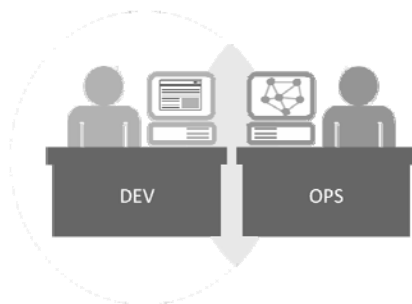


图 1-1 开发工作和运维工作的融合

### 1.1.3 云计算的IaaS

云计算是近年来的热点话题，实际上它仅仅是一种面向服务的理念，它将原本分散在全球各地的 IT 资源集中起来，通过虚拟化、分布式、多租户、自助服务、自动计费的方式递送给用户。云计算很巧妙地将服务模型划分为 IaaS（基础设施即服务）、PaaS（平台即服务）、SaaS（软件即服务）。

IaaS 关注基础架构中最基础的存储、计算、网络三大服务，它的出现很好地解决了中小 IaaS 企业对底层资源管理复杂的问题，人们无须再购买硬件、租赁机房、管理 OS。但 IaaS 的出现对于解决开发、运维工作的困局是远远不够的，在存储、计算、网络之上还有支撑应用运行的各类中间件，除了将存储、计算、网络、中间件等资源绑定成一个整体，还需要对开发代码发布进行严格的安全控制。可以说 IaaS 直接面向的用户是运维人员，而 PaaS 面向的用户是开发人员。

### 1.1.4 PaaS的到来

PaaS 将关注点从原有的基础资源上升到应用层面，它的目标是提供一个可简单操作的平台来帮助开发人员运行、管理 Web 应用，它的关注点围绕着开发者的可运行代码。PaaS 提供一个环节给开发者创建、管理、部署应用，其收益不仅包括了 IaaS 原有的规模效应、固有成本，更看重的是提高代码发布的效率。在资源层面，PaaS 提供底层计算、存储、网络、虚拟化、中间件等服务。在部署上，PaaS 为了黏合开发、运维人员之间的关系，提供了一套自定义的部署工具，工具与企业的适用程度越高，意味着 PaaS 越有可能通过私有云方式提供。除了资源提供、环境部署，具体的 PaaS 甚至会提供团队协作、服务集成、负载均衡、安全控制、持久化、状态管理等类型的服务，随着 PaaS 提供的服务与代码的精密度越来越高，其对应用本身的约束也就越大。这就导致很难有一个标准的公有 PaaS 平台满足绝大多数企业的应用开发需求。

### 1.1.5 PaaS的约束与开放

约束型的 PaaS 平台会高效地利用底层计算、存储、网络资源，但对于开发者来说，他们不得不遵循一套独立的规范和 API（应用程序接口），甚至连基本的开发语言也要随着平台而改变，早期的 PaaS 基本上属于约束型。Google App Engine（GAE）是谷歌公司提供的 PaaS 平台，它承诺基于 GAE 的应用可以享受到谷歌云平台的所有资源优势，并提供了一套行为规范及 API 让开发人员使用，该平台支持的语言有 Python、Java、Go、PHP 语言。GAE 虽然提供了一个强大的基础平台，但有太多的约束性，包括语言库文件受限、服务调用接口为 HTTP、数据检索过滤对象唯一、会话状态持久性等。

开放型的 PaaS 平台不再对开发者的代码进行侵入式的约束，而是提供足够的自由，开发人员保留其原有编程语言、框架和容器组件。开放型 PaaS 还会逐步地与版本控制、持续集成、测试等开发过程工具进行集成。开放型 PaaS 是新一代 PaaS 的尝试。在开放型 PaaS 中，Heroku 可以作为代表，该平台的建设自 2007 年 6 月开发，当时它仅支持 Ruby，但后来增加了对 Java、Node.js、Scala、Clojure、Python，以及（未记录在正式文件上）PHP 和 Perl 的支持。值得一提的是在 2011 年 Ruby 编程语言的设计者松本行弘作为首席架构师加入该平台，对 Heroku 的快速发展发挥了重要作用。

### 1.1.6 PaaS解决的具体问题

既然说 PaaS 要彻底地填补开发、运维工作之间的沟壑，让开发的全部精力聚焦到业务逻辑上，那么我们有必要让 PaaS 解决的问题具体化。

1) PaaS 提供的是一个应用的聚合，这里包含了功能各异的服务组件

- 应用服务中间件：直接包含业务逻辑代码、模块的中间件容器，提供数据库连接池、事务控制等接口以掩盖后端的复杂性。
- 数据存储服务：业务数据的存储区域通过标准的数据存储协议如文档型、SQL、key-value 等交互。
- 消息服务：为了对应用组件间进行解耦，常常需要支持点对点、发布-订阅的消息服务。

2) PaaS 要提供服务发现、可伸缩性、状态管理功能

- 服务发现：组件与组件之间如何查找、发现对方，如何将最新的地址信息通知到应用聚合，如何对外暴露统一的访问点，这是 PaaS 要考虑的一个功能点，具体的实现包括可编程的 DNS 服务器及 IP 注册分配器。
- 可伸缩性：涉及如何快速地对应用进行扩容，组件如何依据类型请求负载的分配及分配的基本机制。
- 状态管理：对于可快速复制、易扩容的组件，如何管理它们的会话状态。

3) PaaS 中的服务监控、恢复与容灾

这是指对于应用聚合中的每一个组件，如何做到简单、自定义地监控，并且在服务异常时启动服务的快速恢复功能。容灾指跨数据中心的平台级故障恢复，涉及两个数据中心之间的逻辑计算单元如何保持通信，如何保持唯一性，业务数据如何进行备份。

4) PaaS 的 Portal 门户

PaaS 提供了一个对用户友好的 Portal，可以基于 UI 进行应用资源的聚合，并且可以快

速地查找到配置信息、计费信息。

### 5) ITIL 服务管理的相关内容

ITIL (Information Technology Infrastructure Library, 即信息技术基础架构库) 由英国政府部门 CCTA (Central Computing and Telecommunications Agency) 在 20 世纪 80 年代末期制定, 现由英国商务部 OGC (Office of Government Commerce) 负责管理, 主要适用于 IT 服务管理 (ITSM)。ITIL 为企业的 IT 服务管理实践提供了一个客观、严谨、可量化的标准和规范。

在云计算之前, 许多大中型企业的 IT 管理方式是基于 ITIL 管理方法论的, 它们制订了一些适用于业务与 IT 服务稳定而快速交付的具体流程、工具和方法, 但随着云平台、PaaS 的出现, ITIL 是否就没有必要存在了呢? 笔者认为对于金融、保险行业生产环境的服务, 如果能够将 ITIL 管理中的控制规则同样通过自定义的方式集成到 PaaS 平台, 那么将会提高服务的可用性。

### 6) PaaS 平台的安全管控

PaaS 平台的安全管控包括三方面: PaaS 平台的组成组件自身的安全控制; PaaS 中提供的服务的安全控制; PaaS 对外部提供服务的统一出口的安全控制。

### 7) 部署发布的相关内容

最后开发的代码如何通过工具自动、快速地发布到平台也是要考虑的, 这部分与开发过程相关, 包括代码单元测试、集成测试、打包、版本控制、部署等。

## 1.2 什么是分布式计算

### 1.2.1 分布式计算与PaaS

分布式计算是专门研究分布式系统的计算机科学领域。一个分布式系统由若干组件组成, 这些组件通过在网络上传递消息来进行通信与协调, 从而完成一个统一的任务。分布式计算研究如何将分散的计算、存储、网络资源集合起来形成一个巨大的计算系统, 该系统可以抽象成一个本地操作系统, 它有强大的计算能力及海量存储空间, 计算任务之间的通信如同在本地一样简单、透明。

广义上的分布式系统无处不在, 只要一个业务操作多个组件协作完成任务, 组件之间跨网络进行通信, 则可以认为这是一个分布式系统。狭义上的分布式系统则完整地定义了一系列组件来形成一致、整体的标准, 包括抽象原型、组合方式、通信协议等。

PaaS 平台实际上是一个完整的分布式系统, 它必须将独立的计算资源组合起来形成一

个抽象的大计算系统，例如如何将数据中心的所有操作系统聚合起来形成一个大操作系统，透明地为各类进程提供计算、存储、网络资源。在 IaaS 层面上通过虚拟化、VPC，以及服务接口方式将基础资源转变成可编程化控制。与 PaaS 相比，IaaS 的分布式不够彻底，因为我们看到的还是独立的操作系统资源，而没有将其抽象到一个独立的 OS 程度。

我们需要讨论分布式计算的原因是 PaaS 平台将作为一个全局性资源出现，PaaS 对外表现为只有一个操作系统、一个文件系统，在这个全局性资源里，我们要依据分布式的原理来拆分计算任务、定义统一服务接口，处理并发下的共享资源互斥、调度可用资源等。

### 1.2.2 分布式平台的挑战

#### 1. 可扩展性

在分布式平台上运行着如下异构的基础资源：网络、服务器硬件、操作系统、中间件、编程语言等。可扩展性，在英文上用 Extensibility 表示。PaaS 平台是否可以做到非常简单地集成、兼容各类基础资源，PaaS 平台是否可以快速加入新的中间件服务，应用是否可以由各类中间件组件组合而成，这是 PaaS 在可扩展性上要考虑的。

网络的可扩展性解决方式通过通用网络协议来掩盖底层的差异，在一个局域网（子网）内采用一致的网络设备进行通信，多个子网间的底层可以采用完全不同的硬件设备，例如传输层采用双绞线、光纤等，在物理层向上的操作系统、网络设备转发数据时，遵循通信协议来解决问题。

操作系统直接安装在服务器硬件上，操作系统本身对硬件服务器的异构进行掩盖，底层很可能采用 32 位或 64 位的 CPU，也可能采用 Intel 或 AMD CPU，指令集上的差异在操作系统层面解决，操作系统对应用程序提供一致的 API，以便于一套代码可以在不同的服务器硬件上运行；代码意指编译前的程序，若采用编译后的程序直接在不同平台上运行，则或多或少会出现问题，编译后的机器码是直接 CPU 硬件平台的指令集对应的。

虚拟机的出现不仅仅尽可能地解决了一套硬件服务器运行多个操作系统的问题，同时实现了一次编译多次运行。VMware、KVM 虚拟机技术使我们能够在服务器层面虚拟多个客户操作系统，基于解释性的编程语言（Java）等在 JVM 中运行，实现了程序快速移植。

在 PaaS 平台上解决可扩展性的前提是在有限的范围内（如一个层级内）统一标准，硬件平台层级是否可以统一 CPU 位级，操作系统层级是否可以全部使用类 Linux 操作系统，计算机的组成如同洋葱一样，层层向上，越靠近应用其种类越多，其标准也就越难控制。

我们难以控制操作系统以上的标准，需要通过自动化脚本来实现安装、部署，另外如果能够提供给用户自助化的组件定义、打包并复制使用，即实现了标准的可变性，那么便将标准的制订交给了开发人员。Docker 是一个可以实现这一功能的应用容器引擎，它让开发者可以打包其应用及依赖包到一个可移植的容器中，然后发布到任意流行的 Linux 机器上，

也可以实现虚拟化。Docker 相当于一个轻量级虚拟机，它给 PaaS 平台带来的最大收益是将种类多样、组合复杂的计算单元打包，形成标准。

统一的通信协议能够掩盖底层的复杂性。早期的 RPC、RMI 协议从编码角度出发，通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议，但由于是从编码层面设计的，通信的双方被绑定在了特定的中间件组件之上。很快，行业内部为了彻底解决中间件的异构问题，发布了基于 HTTP 和 XML 的 SOAP（Simple Object Access Protocol，简单对象访问协议，是交换数据的一种协议规范，使用在计算机 Web 服务中）。实践是检验一切真理的标准，历史的车轮滚滚向前，SOAP 协议的复杂性被证明不适应互联网间的数据通信与交互，而 RESTful 与 JSON 的结合成为了互联网应用 API 的新潮流。

### 2. 可伸缩性

我们首先来区分下可伸缩性（Scalability）与可扩展性（Extensibility）。可扩展性主要指一个功能的扩展，可以认为是功能的增强，扩展性越好的系统，在加入新功能时其代码的修改、架构的变动越少。而可伸缩性指容量的支撑，对于同一功能可以支撑 100 个用户的并发请求，当并发量达到 1 万时，系统如何做到快速扩容以保证稳定属于可伸缩性的范畴。

可伸缩性的解决方法分为两步：第 1 步是资源透明扩容，例如我们如何实现一个硬盘从 1GB 扩容到 1TB，而资源的使用用户对此无从感知；第 2 步是计算任务的分解，例如我们如何将海量的用户请求分发到不同的应用服务上，如何将海量计算的检索查询任务分解成细粒度的计算。这些都是可伸缩性要考虑解决的。

### 3. 容错性

容错性是分布式 PaaS 平台必须考虑的问题，例如我们如何处理 PaaS 平台中各类组件、服务的异常情况，以及 PaaS 平台自身如何容灾。

一般来说，PaaS 平台应当包含一个监控模块，从简单性上看提供一般的 TCP、HTTP 监控即可，但生产环境需要的不仅仅是粗粒度的监控，还需要收集 JVM 应用性能数据、主机数据、网络设备，等等，这就要求我们掌握 JMX、SNMP 等网络通信协议，并对各组件厂商提供的监控数据字典了如指掌，例如对于 WebLogic 我们要了解其主要的 MBean 信息，而对于网络设备我们要找到基于 SNMP 的 OID 信息。

平台级别的容灾应该是跨数据中心的，如果以数据中心定义平台单位，那么如何为每个单位划定物理资源、共享平台配置信息、应用配置信息以实现快速容灾是我们要讨论的另一个话题。

### 4. 安全性

由于在分布式平台上运行的应用系统涉及业务数据，组件维护与通信常常面向业务数据，因此对平台的安全性设计应当重点考虑。安全性主要体现在两个方面：一是数据的保密性，涉及如何定义数据的访问权限，如何保护敏感信息不被非授权人员查看；二是服务

的可用性，例如如何防止外部攻击破坏正常的服务可用性。

### 5. 并发性

并发性问题在单一操作系统组件上也会出现，例如数据库对某一全局数据的更新操作。最简单的方式是通过锁来解决共享资源的互斥访问。单一组件对于并发的两个请求客户端来说，它们之间更加容易达成一致，从而实现锁机制。分布式平台可能面临着高并发访问、海量数据检索。而客户端之间的通信与单一组件相比将更加复杂，如何形成统一标准来管理并发性问题是 PaaS 平台设计之初就必须考虑的，这里涉及组件之间如何通信、协调并且达成一致。

# PaaS 模型与特征

## 2.1 主流PaaS平台架构

公有 PaaS 平台并没有达成共识，没有统一应用的 PaaS 服务 API，因此不便于应用在各平台之间移植。谷歌、亚马逊与微软三大巨头在 PaaS 领域分庭对立，在强大的技术实力与基础资源的支撑下，构建了与自身文化相对应的公有云 PaaS 平台。相对于三大巨头，于 2007 年起家的 Heroku，正是由于看到了大平台厂商对应用代码的“侵入性”，以及对开发人员的“绑架”，因而独辟蹊径地开发了一套可移植的 PaaS 平台。

除了用户可直接使用的公有云资源，商用的私有 PaaS 软件与解决方案也受到了企业 IT 用户的追捧，其最大的优势是按照企业客户的要求定制化。老牌企业级 Linux 服务提供商 RedHat 公司也加入了这场私有 PaaS 市场份额争夺战，其产品依赖开源软件构件而成，利用其原操作系统产品在企业 IT 中的广泛应用，其运维支持团队长期深入企业 IT 一线，非常熟悉企业用户的痛点。另外要提及的一个企业级 PaaS 平台是由 VMware 公司构件的 Cloud Foundry，它是业界第一个开源云平台。

### 2.1.1 谷歌GAE

GAE (Google App Engine) 可让你利用谷歌的基础设施构建和运行应用程序。基于 GAE 构建的应用程序能够非常容易地应对访问量、存储空间的变化。GAE 支持的编程语言包括 Java、Python、PHP、Go。它包括以下特性：

- 具有查询、排序与事物控制的持久化存储；
- 自动扩展和负载均衡；
- 用了执行额外任务的异步消息队列；
- 按照指定时间与规则执行任务的事件触发器；
- 可与其他谷歌云服务和 API 集成。



开发人员利用 GAE 简化了 Web 应用程序的开发和部署。图 2-1 是 GAE 上的 Web 架构简图，在这个架构中应用程序可以使用自动伸缩计算的资源，同时可集成分布式缓存、任务队列、数据存储等服务。GAE 有自己的云平台 SDK 库，使应用程序能快速地部署和运行到云上。

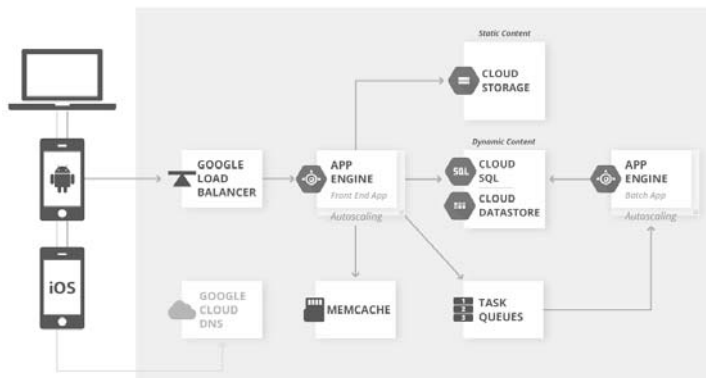


图 2-1 GAE 上的 Web 架构简图

在这个架构下应用流量可被路由到多个版本以支持 A/B 测试。App Engine 相当于计算资源，它分为 service（面向用户）和 batch（后台任务）两类。AppEngine Memcache 在架构中是一个内存共享实例，充当缓存使用，我们可以将身份验证、会话信息等存放在这里来提升 Web 服务器性能。Task queues 提供了一种机制，将需要后端计算资源的任务保存到队列中继续等待，释放了前端在这些任务上的阻塞 I/O、连接，从而持续地为新用户请求提供服务。其负载均衡器支持网络的 3~7 层。DNS 服务可以用来管理自己的整个 DNS zone。

## 2.1.2 AEB

AEB（AWS Elastic Beanstalk）提供了一套在亚马逊云上部署与管理应用的简单方法。用户可以简单地上传应用程序包，AEB 会对应用程序包自动进行容量评估、负载均衡、自动伸缩及健康检查。如图 2-2 所示。

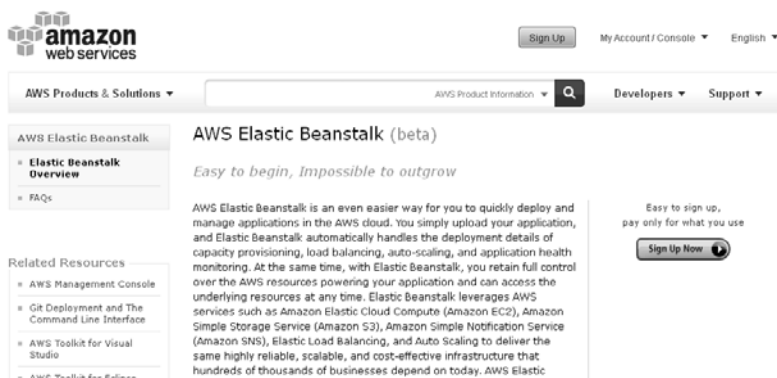


图 2-2 AEB 界面

AEB 的组件包括如下几种。

### 1) Application

Application 是组件的逻辑集合，它包括了后面提到的 Environment、Version 和 Environment Configuration，可以将一个 Application 看作一个目录。

### 2) Version

在 AEB 中，Version 代表一个 Web 应用的特定代码版本，它指向了亚马逊简单的存储服务上的一个对象，一般包含了可部署代码，比如 Java 的 war 包。应用可以包含多个 Version，这些可部署代码由用户上传并打上了版本标签。在亚马逊云上，你可以在多个 Version 间切换，以测试、验证版本间的不同。Version 存放在分布式对象存储区中。

### 3) Environment

Environment 是部署在 AWS 平台上的一个可运行的 Version，每一个 Environment 在一个时间点上只能运行一个 Version，但是你可以同时启动多个包含不同 Version 的 Environment，以测试它们之间的差异。在创建 Environment 的时候，AEB 就自动将资源分配给了特定的 Version。

AEB 的 Environment 有两种类型，一种是提供 HTTP 请求的 Web 服务，另一种是后台任务，这是依据分布式计算模型对 Environment 进行的划分，后面我们还会详细讲解这两种计算模型，以及不同的分布式处理方法。在 AEB 中，前者被命名为 Web Server Environment，后者被命名为 Worker Environment。

在 Web Server Environment Tier 架构中，Environment 是应用的核心。Web Server Environment 的架构示例如图 2-3 所示。在创建一个 Environment 时，AWS Elastic Beanstalk 规定了运行应用所需的资源，如图 2-3 所示的资源包括负载均衡器（Elastic Load Balancer）、一个自动伸缩功能组和多个 Amazon EC2 实例。

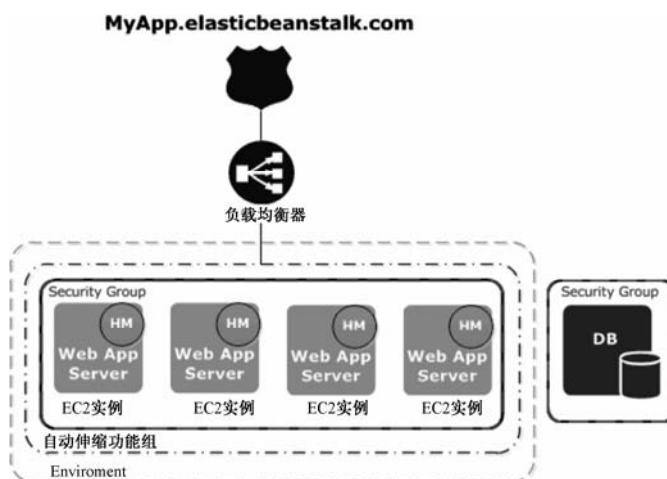


图 2-3 Web Server Environment 架构

每个 Enviroment 的访问入口是一个 CNAME 域名，它被路由到负载均衡器的 IP 地址。图中的域名是 MyApp.elasticbeanstalk.com。在亚马逊云边界的最外端有一个功能强大的 DNS 服务器，它会接收用户的域名查询工作，并将后端配置在负载均衡上的正常的服务 IP 返回给用户，在这里它提供了安全可靠的路由功能。负载均衡的后面是一组 Amazon EC2 实例，它们组成了一个自动伸缩功能组。自动伸缩功能将自动依据当前的负载情况启动冗余的 EC2 实例。随着负载的减少，自动伸缩功能会减少实例，但它会保持一个最小运行实例数目。

HM (Host Manager) 是一个运行态的容器，在这个容器中包含了由用户定义的一组软件栈，例如我们定义了一个 Apache Tomcat 容器，这个容器使用 RedHat Linux 作为操作系统，安装了 Apache HTTPD 服务器和 Tomcat Java 应用服务器。每一个 Amazon EC2 相当于一个计算单元。

Security Group 为运行的 EC2 实例定义了防火墙策略，在默认情况下，AEB 只运行用户访问实例的 80 (HTTP) 端口，你可以依据业务类型定义更多的策略。

### 2.1.3 Cloud Foundry

Cloud Foundry 是由 VMware 贡献的一个开源 PaaS 项目，如图 2-4 所示，它是一个基于 Ruby on Rails 的由多个相对独立的子系统通过消息机制组成的分布式系统，支持多种框架、语言、运行时环境、云平台及应用服务，使开发人员能够在几秒内进行应用程序的部署和扩展。它是一个开源项目，没有专门的公有云环境可供使用，不像 GAE、AWS 的 PaaS 只需要关注应用代码，Cloud Foundry 需要企业的 IT 人员在自己的 IDC 或公有 IaaS 上构建一个私有 PaaS 平台。Cloud Foundry 的创新点在于使用了一种全新的部署代码的方式。针对部署工作，它定义了一套 REST API，底层基于 Ruby 命令行工具来与版本控制器交互，在这个平台上你可以使用 CVS、Subversion、Git 等各种版本的控制器，而不是仅限其一。

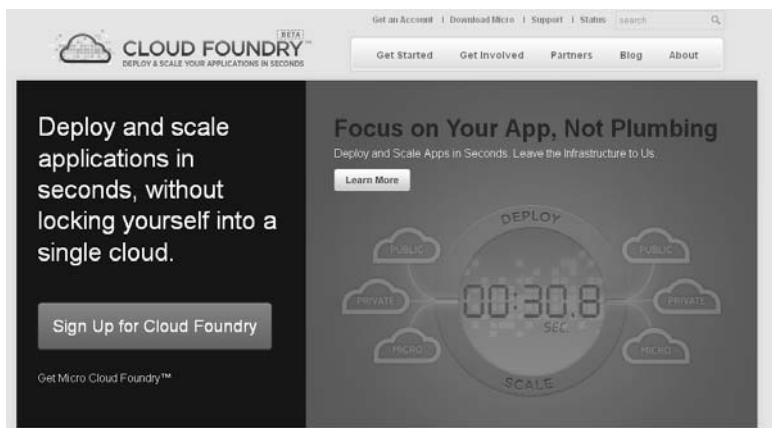


图 2-4 VMware 的 Cloud Foundry

## 2.1.4 Heroku

Heroku 是一个支持多种编程语言的公有 PaaS 平台，其成立于 2007 年，3 年后被 Salesforce.com 收购。Heroku 作为最初的云平台之一，支持 Ruby、Java、Node.js、Scala、Clojure、Python 等多种编程语言。基础操作系统是 Debian，最新的堆栈则是基于 Debian 的 Ubuntu。

Heroku 的架构简图如图 2-5 所示，Heroku 的容器单元被称为 dyno，dyno 越多，应用系统就拥有越多的实例来保证其服务的有效性。

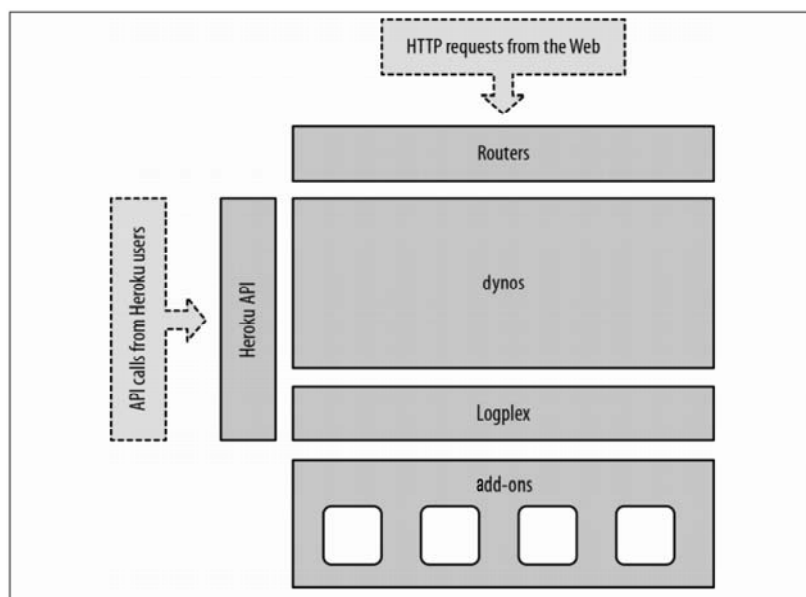


图 2-5 Heroku 的架构简图

Heroku 的路由模块被称为 Hermes，采用 Erlang 语言编写，其能够动态感知一个应用中包含多少个 dyno，基于一定的策略进行任务分发，另外我们还可以设置超时保护机制，在 Hermes 上就拒绝掉外部请求。

Heroku 打破了日志输出的传统观点，我们一般认为日志是非常重要、不可缺失的，日志以文件的形式存放在本地磁盘中，并且有开头、结尾，重视日志文件中每一行内容在时间排序上的关联性。而 Heroku 将日志看作一条一条的流式信息，它将这些输出发送到远端，集中管理、预警。

一个 PaaS 平台会提供大量的后端服务组件，包括持久化数据库、邮件 SMTP 服务、消息队列、缓存等。Heroku 就为这些后端服务的访问定义了一套 add-ons API，从而实现了代码与某个固定服务的解耦。在 Heroku 上最流行的后端服务是 PostgreSQL 数据库。

## 2.2 PaaS与 12-Factor

传统应用会因一些自身问题导致难以伸缩扩展、快速迁移，例如日志输出本地、会话状态记录等，这些问题解决后，一个应用能够享受到 PaaS 的所有好处。当应用与 PaaS 紧密结合，功能服务具有通用性且直接面向用户时，这个应用会演变成 SaaS。Heroku 开发团队在收集与观察了大量运行在 PaaS 上的应用程序的开发、执行过程之后，整理出 12 条开发 SaaS 应用程序的规则，也是最适合 PaaS 的应用程序方法论，统称为 12-Factor，其目标是：

- 使用标准化流程自动配置，从而使新的开发者花费最少的学习成本加入这个项目；
- 和操作系统之间尽可能地划清界限，在各个系统中提供最大的可移植性；
- 适合部署在现代的云计算平台上，从而在服务器和系统管理方面节省资源；
- 将开发环境和生产环境的差异降至最低，并使用持续交付方式实施敏捷开发；
- 可以在工具、架构和开发流程不发生明显变化的前提下实现扩展。

这套方法论适用于任意语言和后端服务（数据库、消息队列、缓存等）开发的应用程序，下面是对 12-Factor 规则的详细说明。

### 2.2.1 基准代码（Codebase）

应用代码使用版本控制器进行管理，这些版本控制器包括 Git、Mercurial、Subversion 等。在控制器中用来跟踪代码所有修订版本的数据库被称作代码库。在类似 SVN 这样的集中式版本控制系统中，基准代码就是指控制系统中的代码库；而在像 Git 那样的分布式版本控制系统中，基准代码则是指最上游的代码库。

基准代码的多份部署（Deploy）如图 2-6 所示。

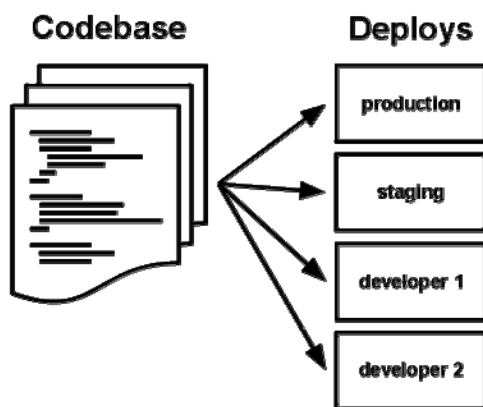


图 2-6 基准代码的多份部署

基准代码和应用之间要求保持一一对应的关系：

- 一旦有多个基准代码，就不能称为一个应用，而是一个分布式系统。分布式系统中的每一个组件都是一个应用，每一个应用可以分别使用 12-Factor 进行开发；
- 多个应用共享一份基准代码是有悖于 12-Factor 原则的。解决方案是将共享的代码拆分为独立的类库，然后使用依赖管理策略去加载它们。

尽管每个应用只对应一份基准代码，但可以同时存在多份部署。每份部署相当于运行了一个应用的实例。通常会有一个生产环境、一个或多个预发布环境。此外，每个开发人员都会在自己的本地环境中运行一个应用实例，这些都相当于一份部署。

所有部署的基准代码相同，但每份部署可以使用不同的版本或者分支。比如，开发人员可能有一些提交还没有同步至预发布环境，预发布环境也有一些提交没有同步至生产环境，但都共享一份基准代码，我们就认为这只是相同应用的不同部署而已。

从运维人员的角度来看，他们交付给开发人员的是可以运行应用程序的标准环境，例如一个安装了各类工具、软件的操作系统。运维人员与开发需要交互的地方在于如何将应用逻辑放入这个环境中。从应用代码到可执行的应用逻辑需经过版本选择、代码编译、配置修改、版本构建、应用发布等过程。12-Factor 从静态代码开始进行规范控制，遵循了该基准代码的应用，在构建一个独立的可执行环境时，可以自动地在可执行环境中下载应用代码，进入后续步骤，而无须再在运维人员与开发人员之间交互。

### 2.2.2 依赖（Dependency）

我们常常会听到开发人员向运维人员抱怨：“请保证环境的一致性”，但实际情况却总是代码或者应用逻辑在新基础环境中出现问题。这是由于应用代码的编译、执行不仅仅与操作系统、中间件相关，还直接依赖于第三方库文件。这些库文件依据编程语言可分为很多种类，例如以 .so 为后缀的动态链接库、以 .jar 为后缀的 Java 库文件等。当应用的基础环境变更而这些库文件版本又与原来所需不一致时，则会导致异常。

在 Linux 操作系统上都有默认的包管理系统，例如 RedHat 的 yum、Ubuntu 的 apt-get。不同的编程语言也有类似的包管理系统，例如 Perl 的 CPAN、Ruby 的 Rubygems，以及 Python 的 Pip。在进行第三方库安装时，库文件可以是系统级别的，安装在 Linux 操作系统的 /usr/lib 目录下，供所有程序引用；也可以放在应用自己的部署目录中，通过指定环境变量来引用。

12-Factor 规范下的应用程序不会隐式依赖系统级的类库。它一定通过依赖清单，确切地声明所有依赖项。此外，在运行过程中通过依赖隔离工具来确保程序不会调用系统中存在但清单中未声明的依赖项。这一做法会统一应用到生产环境和开发环境中。

依赖清单在声明所需库文件的同时，可直接从打包管理系统中下载安装，但有一种情况是对于系统级的库文件，多个应用程序可能同时引用了不同的版本，依赖隔离工具相当于给应用程序一个独立的环境变量来执行上下文，将它所需要引用的第三方库路径从默认的系统级别路径指定到应用路径。

例如，Ruby 的 Gem Bundler 使用 Gemfile 作为依赖项声明清单，使用 bundle exec 来进行依赖隔离。在 Python 中则可分别使用两种工具：Pip 用作依赖声明，Virtualenv 用作依赖隔离。甚至 C 语言也有类似的工具：Autoconf 用作依赖声明，静态链接库用作依赖隔离。对于 Java 应用，在移交运行环境时，使用 Maven、Ant 等工程构建工具将依赖库文件打包到一起，这就是一个清晰的依赖清单了。依赖声明和依赖隔离必须一起使用，否则无法满足 12-Factor 规范。

12-Factor 应用同样不会隐式地依赖某些系统工具，如 ImageMagick 或者 curl。即使这些工具存在于几乎所有系统中，也无法保证未来的所有系统都能支持应用的顺利运行，或者和应用兼容。如果应用必须使用到某些系统工具，那么这些工具应该被包含在应用之中。

显式声明依赖的优点之一是为新加入的开发者简化了环境配置流程。这些开发者可以检测出应用程序的基准代码，安装编程语言环境和与它对应的依赖管理工具，只需通过一个构建来安装所有的依赖项，即可开始工作。例如，在 Ruby/Bundler 下使用的是 bundle install，而在 Clojure/Leiningen 下则是 lein deps。依赖隔离用来在同一个操作系统上运行互不影响同类应用程序。

12-Factor 的依赖是在每次 checkout 一份代码时再构建环境，这是“事后”的构建方法。基于 Puppet、Saltstack 软件状态配置工具，按照 DSL 依赖清单来构建的方法与其类似。在 PaaS 平台上还可以采用基于“事前”的方法，即提前将应用所依赖的基础环境作为镜像构建好，基于 Docker 的 AUFS 分层方式是一种镜像实现，这样只要使用的是同一份镜像，即可保持环境的一致性。

### 2.2.3 配置 (Config)

配置是指将配置信息存储在环境变量中。基础资源（操作系统、中间件、库文件）、应用代码、应用逻辑在多套部署环境（预发布、生产环境、开发环境等）中是可以完全一致的。而各套环境间的最终差异体现在应用与外部组件的通信上，这些通信内容包括：

- 数据库，如 Memcached 及其他后端服务的配置；
- 第三方服务的证书，如 Amazon S3、Twitter 等；
- 每份部署特有的配置，如域名等。

这些差异会反映在配置中。有些应用在代码中使用常量保存配置，这与 12-Factor 所要

求的“代码与配置严格分离”显然大相径庭。配置文件在各部署之间存在大幅差异，代码却完全一致。

判断一个应用是否正确地将配置排除在代码之外的一种简单方法是看该应用的基准代码是否可以立刻开源，而不用担心是否暴露任何敏感的信息。

需要指出的是，这里定义的“配置”并不包括应用的内部配置，比如 Rails 的 `config/routes.rb`，或者使用 Spring 时代码模块间的依赖注入关系。这类配置在不同部署间不存在差异，所以应该写入代码。

另外一种解决方法是使用配置文件，但不把它们纳入版本控制系统，就像 Rails 的 `config/database.yml`。这相对于在代码中使用常量已经是很大的进步了，但仍然有缺陷：总是会一不小心将配置文件嵌入了代码库；配置文件可能会分散在不同的目录中，并有着不同的格式，这使找出一个空间来统一管理所有配置变得不太现实。更糟糕的是，这些格式通常是语言或框架特定的。

12-Factor 推荐将应用的配置存储于环境变量中（如 `env vars`、`env`）。环境变量可以非常方便地在不同的部署间做修改，却不动一行代码；与配置文件不同，一不小心把它们嵌入代码库的概率微乎其微；与一些传统的解决配置问题的机制（比如 Java 的属性配置文件）相比，环境变量与语言、系统无关。

配置管理的另一个方面是分组。有时应用会将配置按照特定部署进行分组（或叫作“环境”），例如 Rails 中的 `development`、`test` 和 `production` 环境。这种方法无法轻易扩展：更多部署意味着更多新的环境，例如 `staging` 或 `qa`。随着项目的不断深入，开发人员可能还会添加它们自己的环境，比如 `joes-staging`，这将导致各种配置组合的激增，从而给管理部署增加了很多不确定因素。

在 12-Factor 应用中，环境变量的粒度要足够小且相对独立。它们永远也不会组合成一个所谓的“环境”，而是独立存在于每个部署之中。当应用程序不断扩展，需要更多种类的部署时，这种配置管理方式能够做到平滑过渡。

### 2.2.4 后端服务（Backing Services）

#### 1. 把后端服务当作附加资源

后端服务是指程序运行所需要的通过网络调用的各种服务，如数据库（例如 MySQL、CouchDB）、消息/队列系统（例如 RabbitMQ、Beanstalkd）、SMTP 邮件发送服务（例如 Postfix），以及缓存系统（例如 Memcached）。

类似于数据库的后端服务，通常由部署应用程序的系统管理员一起管理。除了本地服务，应用程序有可能使用了第三方发布和管理的服务。示例包括 SMTP（例如 Postmark）、



数据收集服务（例如 New Relic 或 Loggly）、数据存储服务（例如 Amazon S3），以及使用 API 访问的服务（例如 Twitter、Google Maps、Last.fm）。

## 2. 12-Factor 应用不会区别对待本地或第三方服务

对应用程序而言，本地都是附加资源，通过一个 URL 或者其他存储在配置中的服务定位/服务证书来获取数据。12-Factor 应用的任意部署，都应该可以在不进行任何代码改动的前提下，将本地 MySQL 数据库转换成第三方服务（例如 Amazon RDS）。类似地，本地 SMTP 服务应该也可以和第三方 SMTP 服务（例如 Postmark）互相转换。在上述两种转换中，仅需修改配置中的资源地址。

每个不同的后端服务都是一个资源。例如，一个 MySQL 数据库是一个资源，两个 MySQL 数据库（用来数据分区）被当作两个不同的资源。12-Factor 应用将这些数据库都视作附加资源，这些资源和它们附属的部署保持松耦合，如图 2-7 所示。

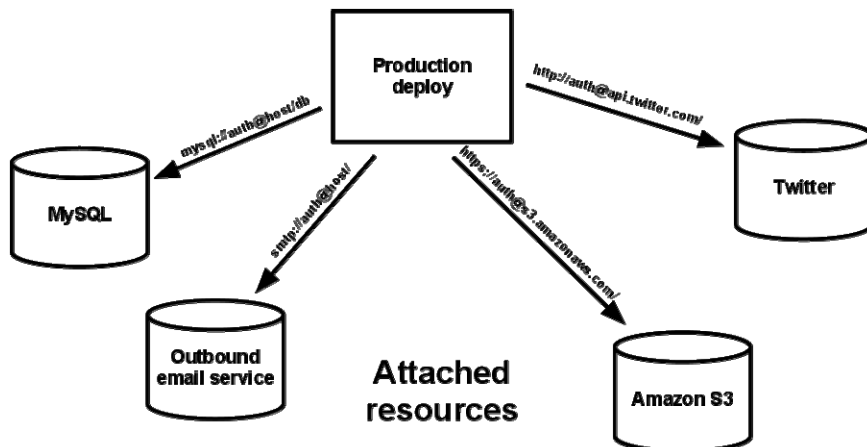


图 2-7 附加资源间的松耦合

部署可以按需加载或卸载资源。例如，如果应用的数据库服务由于硬件问题出现异常，那么管理员可以从最近的备份中恢复一个数据库，卸载当前的数据库，然后加载新的数据库，在整个过程中都不需要修改代码。

### 2.2.5 构建（Build）、发布（Release）、运行（Run）

基准代码转化为一份部署（在非开发环境下）需要以下三个阶段，如图 2-8 所示。

- 构建阶段是指将代码仓库转化为可执行包的过程。在构建时会使用指定版本的代码来获取和打包依赖项，编译成二进制文件和资源文件；
- 发布阶段会将构建的结果和当前部署所需的配置相结合，并能够立刻在运行环境中投入使用；

- 运行阶段（或者说“运行时”）指针对选定的发布版本，在执行环境中启动一系列应用程序的进程。

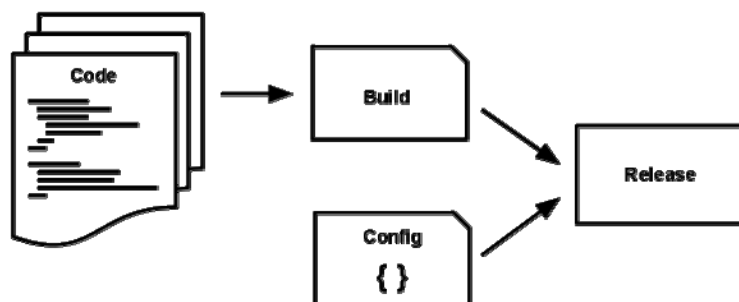


图 2-8 构建-发布-运行

12-Factor 应用严格区分构建、发布、运行这三个步骤。举例来说，直接修改处于运行状态的代码是非常不可取的做法，因为这些修改很难再同步到构建步骤。

部署工具通常都提供了发布管理工具，最引人注目的功能是退回至较旧的发布版本。比如，**Capistrano** 将所有发布版本都存储在一个叫作 **releases** 的子目录中，当前的在线版本只需映射至对应的目录即可。该工具的 **rollback** 命令可以很容易地实现回退版本的功能。

每一个发布版本必须对应一个唯一的发布 ID，例如可以使用发布时的时间戳（2011-04-06-20: 32: 17），或者一个增长的数字（v100）。要发布的版本就像一本只能追加的账本，一旦发布就不可修改，任何变动都应该产生一个新的发布版本。

新的代码在部署之前，需要开发人员触发构建操作。但是，在运行阶段不一定需要人为触发，可以自动进行，例如服务器重启，或者进程管理器重启了一个崩溃的进程。因此，在运行阶段应该保持尽可能少的模块，假设半夜发生系统故障而开发人员又捉襟见肘也不会有太大的问题。在构建阶段可以相对复杂一些，因为错误的信息能够立刻展示在开发人员面前，从而得到妥善处理。

## 2.2.6 进程（Process）

在运行环境中，应用程序通常是以一个或多个无状态进程运行的。

在最简单的场景中，代码是一个独立的脚本，运行环境是开发人员自己的笔记本电脑，进程命令是一条 **Shell** 语句（例如 **Python my\_script.py**）。另外一种极端情况是，复杂的应用可能会使用很多进程类型，也就是零个或多个进程实例。

12-Factor 应用的进程必须无状态且无共享。任何需要持久化使用的数据都要存储在后端服务内，比如数据库。

内存区域或磁盘空间可以作为进程在做某种事务型操作时的缓存，例如下载一个很大

的文件，对其进行操作并将结果写入数据库。12-Factor 应用根本不用考虑这些缓存的内容是不是可以留给之后的请求使用，这是因为应用启动了多种类型的进程，将来的请求多半会由其他进程来服务。即使在只有一个进程的情形下，先前保存的数据（在内存或文件系统中）也会因为重启（如代码部署、配置更改或运行环境将进程调度至另一个物理区域执行）而丢失。

源文件打包工具（Jammit、django-assetpackager）使用文件系统来缓存编译过的源文件。12-Factor 应用更倾向于在构建阶段进行缓存，例如在 Rails 资源管道阶段，而不是运行阶段。

一些互联网系统依赖于“黏性 session”，这是指将用户 session 中的数据缓存至某进程的内存中，并将同一用户的后续请求路由到同一个进程中。黏性 session 是 12-Factor 极力反对的。Session 中的数据应该保存在诸如 Memcached 或 Redis 这样的带有过期时间的缓存中。

## 2.2.7 端口绑定（Port Binding）

互联网应用有时会运行于服务器的容器之中。例如 PHP 经常作为 Apache HTTPD 的一个模块来运行，正如 Java 运行于 Tomcat 上。

12-Factor 应用完全自我加载而不依赖于任何网络服务器就可以创建一个面向网络的服务。互联网应用通过端口绑定来提供服务，并监听发送至该端口的请求。

在本地环境中，开发人员通过类似 `http://localhost:5000/` 的地址来访问服务。在线上环境中，请求统一发送至公共域名，然后被路由到绑定了端口的网络进程中。

通常的实现思路是，将网络服务器类库通过依赖声明载入应用。例如，基于 Python 的 Tornado、基于 Ruby 的 Thin 和 Java 及其他基于 JVM 语言的 Jetty，完全由用户端（确切地说应该是应用的代码）发起请求，和运行环境约定好绑定的端口即可处理这些请求。

HTTP 并不是唯一一个可以由端口绑定并提供的服务。其实几乎所有服务器软件都可以通过进程绑定端口来等待请求。例如，使用 XMPP 的 ejabberd，以及使用 Redis 协议的 Redis。

还需要指出的是，端口绑定这种方式也意味着一个应用可以成为另外一个应用的后端服务，调用方将服务方提供的相应 URL 当作资源存入配置，以备将来调用。

## 2.2.8 并发（Concurrency）

任何计算机程序一旦启动，就会生成一个或多个进程。互联网应用采用多种进程运行方式。例如，PHP 进程作为 Apache 的子进程存在，随请求按需启动。Java 进程则采用了相反的方式，JVM 在程序启动之初就提供了一个超级进程来储备大量的系统资源（CPU 和内存），并通过多线程实现内部的并发管理。这里的进程是开发人员可以操作的最小单位。

在 12-Factor 应用中，进程是一等公民。12-Factor 应用的进程主要借鉴于 UNIX 守护进程模型（见图 2-9）。开发人员可以运用这个模型去设计应用架构，将不同的工作分配给不同的进程类型。例如，HTTP 请求可以交给 Web 进程来处理，而常驻的后台工作则交给 worker 进程负责。

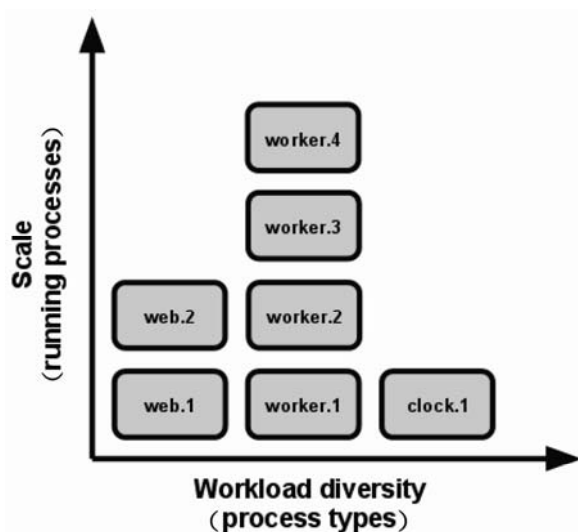


图 2-9 UNIX 守护进程模型

这并不包括个别较为特殊的进程，例如通过虚拟机的线程处理并发的内部运算，或者使用诸如 EventMachine、Twisted、Node.js 的异步/事件触发模型。但一台独立的虚拟机的扩展有瓶颈（垂直扩展），所以应用程序必须可以在多台物理机器间跨进程工作。

上述进程模型会在系统急需扩展时大放异彩。12-Factor 应用的进程所具备的无共享、水平分区的特性意味着添加并发会变得简单而稳妥。这些进程的类型及每个类型中进程的数量就被称作进程构成。

12-Factor 应用的进程不需要守护进程或者写入 PID 文件。相反，应该借助操作系统的进程管理器（例如 Upstart、分布式的进程管理云平台或者类似于 Foreman 的工具）来管理输出流、响应崩溃的进程及处理用户触发的重启和关闭超级进程的请求。

## 2.2.9 快捷性（Disposable）

快速启动和优雅终止可最大化健壮性。

12-Factor 应用的进程是有快捷性的，即它们可以瞬间开启或停止，这有利于快速、弹性地伸缩应用，迅速部署发生变化的代码或配置，稳健地部署应用。

进程应当追求最小启动时间。在理想状态下，进程从发布命令到真正启动并等待请求

的时间应该很短。更少的启动时间提供了更敏捷的发布及扩展过程，此外还增加了健壮性，因为进程管理器可以在被授权的情况下很容易地将进程搬到新的物理机器上。

进程一旦接收终止信号（SIGTERM），就会优雅终止。就网络进程而言，优雅终止指停止监听服务的端口，即拒绝所有新的请求，并继续执行当前已接收的请求，然后退出。此类型的进程所隐含的要求是 HTTP 请求大多都很短（不会超过几秒），而在长时间的轮询过程中，客户端在丢失连接后应该马上尝试重新连接。

对于 worker 进程来说，优雅终止指将当前任务退回队列。例如，在 RabbitMQ 中，worker 可以发送一个 NACK 信号。在 Beanstalkd 中，任务终止并退回队列会在 worker 断开时自动触发。有锁机制的系统诸如 Delayed Job 则需要确定已释放系统资源。此类型的进程所隐含的要求是，任务都应该可重复执行，这主要由将结果包装进事务或者幂等来实现。

进程还应当在面对突然死亡时保持健壮，例如底层硬件故障。虽然这种情况比起优雅终止来说少之又少，但终究有可能发生。一种推荐的方式是使用一个健壮的后端队列如 Beanstalkd，它可以在客户端断开或超时后自动退回任务。无论如何，12-Factor 应用都应该可以设计能够应对意外的、不优雅的终结。Crash-only design 将这种概念转化为合乎逻辑的理论。

### 2.2.10 开发/生产环境等价（Dev/Prod Parity）

应尽可能地保持开发、预发布及线上环境相同。从以往经验来看，开发环境（即开发人员的本地部署）和线上环境（外部用户访问的真实部署）之间存在着很多差距。这些差距表现在以下三个方面。

- 时间差距：开发人员正在编写的代码可能需要几天、几周，甚至几个月才会发布生产。
- 人员差距：开发人员编写代码，运维人员部署代码。
- 工具差距：开发人员可能使用 Nginx、SQLite、OS X，而线上环境使用 Apache、MySQL 及 Linux。

12-Factor 应用要想做到持续部署，就必须缩小本地环境与线上环境的差距。再回头看看上面所描述的三个差距。

- 缩小时间差距：开发人员可以在几小时甚至几分钟内就部署完代码。
- 缩小人员差距：开发人员不仅要编写代码，还要积极参与部署的过程，并关注代码在线上环境中的表现。
- 缩小工具差距：尽量保证开发环境及线上环境的一致性。

上述总结如表 2-1 所示。

表 2-1 开发与运维的差异

对比项目	传统应用	12-Factor 应用
每次部署间隔	数周	几小时
开发人员 vs 运维人员	不同的人	相同的人
开发环境 vs 线上环境	不同	尽量接近

后端服务是保持开发与线上等价的重要部分，例如数据库、队列系统及缓存。许多语言都提供了简化获取后端服务的类库，例如不同类型服务的适配器。下面是一些例子，如表 2-2 所示。

表 2-2 语言类库

类 型	语 言	类 库	适 配 器
数据库	Ruby/Rails	ActiveRecord	MySQL、PostgreSQL、SQLite
队列	Python/Django	Celery	RabbitMQ、Beanstalkd、Redis
缓存	Ruby/Rails	ActiveSupport: Cache	Memory、filesystem、Memcached

开发人员有时会觉得在本地环境中使用轻量的后端服务具有很强的吸引力，而那些更重量级的健壮的后端服务应该使用在生产环境中。例如，本地环境使用 SQLite，线上环境使用 PostgreSQL；又如本地环境缓存在进程内存中，而线上环境缓存在 Memcached 中。

12-Factor 应用的开发人员应该反对在不同的环境中使用不同的后端服务，即使适配器已经几乎可以消除使用上的差异。这是因为，不同的后端服务意味着会突然出现不兼容的状况，从而导致测试、预发布都正常的代码在线上出现问题，这些错误会给持续部署带来阻力。从应用程序的生命周期来看，消除这种阻力需要付出很大的代价。

与此同时，轻量的本地服务也不像以前那样引人注目。借助于 Homebrew、apt-get 等现代打包系统，诸如 Memcached、PostgreSQL、RabbitMQ 等后端服务的安装与运行也并不复杂。此外，使用类似 Chef 和 Puppet 的声明式配置工具，结合像 Vagrant 这样轻量级的虚拟环境就可以使得开发人员的本地环境与线上环境无限接近。与同步环境和持续部署所带来的益处相比，安装这些系统显然是值得的。

不同后端服务的适配器仍然是有用的，因为它们可以使移植后端服务变得简单。但应用的所有部署，包括开发、预发布及线上环境，都应该使用同一个后端服务的相同版本。

### 2.2.11 日志 (Log)

#### 1. 把日志当作事件流

日志使得应用程序运行的动作变得透明。在基于服务器的环境中，日志通常被写在硬

盘的一个文件里，但这只是一种输出格式。

日志应该是事件流的汇总，将所有运行中的进程和后端服务的输出流按照时间顺序收集起来。尽管在回溯问题时可能需要看很多行，但日志最原始的格式确实是一个事件为一行。日志不像文件一样有开头和结尾，而是像流一样随着应用的运行持续增长。

## 2. 12-Factor 应用本身从不考虑存储自己的输出流

不应该试图去写或者管理日志文件。相反，每一个运行的进程都会向标准输出（`stdout`）直接发送事件流。在开发环境中，开发人员可以通过这些数据流，实时地在终端看到应用的活动。

在预发布或线上部署中，每个进程的输出流由运行环境截获，并将其他输出流整理在一起，然后一并发送给一个或多个最终的处理程序，用于查看或者长期存档。这些存档路径对于应用来说既不可见也不可配置，而是完全交给程序的运行环境管理的。类似于 Logplex 和 Fluent 的开源工具可以达到这个目的。

这些事件流可以输出至文件，或者在终端实时观察。最重要的是，输出流可以发送到 Splunk 这样的日志索引及分析系统，或 Hadoop/Hive 这样的通用数据存储系统中。这些系统为查看应用的历史活动提供了强大而灵活的功能，包括：

- 找出过去一段时间内特殊的事件；
- 图形化一个大规模的趋势，比如每分钟请求量；
- 根据用户定义的条件实时触发警报，比如每分钟的报错超过某个警戒线。

### 2.2.12 管理进程（Admin Process）

进程组指用来处理应用的常规业务（比如处理 Web 请求）的一组进程。与此不同，开发人员经常希望执行一些管理或维护应用的一次性任务，例如：

- 运行数据移植（Django 中的 `manage.py migrate`，Rails 中的 `rake db: migrate`）；
- 运行一个控制台（也被称为 REPL Shell），执行一些代码或者针对线上数据库做一些检查。大多数语言（例如 Python）都通过解释器提供了一个 REPL 工具（Python 或 Perl），或者其他命令（Ruby 使用 `irb`，Rails 使用 `rails console`）；
- 运行一些提交到代码仓库的一次性脚本。

一次性管理进程应该和正常的常驻进程一样使用同样的环境。这些管理进程和任何其他进程一样使用相同的代码和配置，基于某个已发布的版本运行。后台管理代码应该随其他应用程序一起发布，从而避免发生不同步问题。

所有进程类型应该使用同样的依赖隔离技术。例如，如果 Ruby 的 Web 进程使用了命令

`bundle exec thin start`，那么数据库移植应使用 `bundle exec rake db: migrate`。同样，如果一个 Python 程序使用了 Virtualenv，则需要在运行 Tornado Web 服务器和任何 `manage.py` 管理进程时引入 `bin/Python`。

12-Factor 尤其青睐那些提供了 REPL Shell 的语言，因为那会让运行一次性脚本变得简单。在本地部署中，开发人员直接在命令行中使用 Shell 命令调用一次性管理进程。在线上部署中，开发人员依旧可以使用 SSH 或者运行环境提供的其他机制来运行这样的进程。

## 2.3 PaaS与Reaction宣言

12-Factor 值得细细品读、深刻体会，并将其应用到工作中。与此同时，另一份称为“Reaction 宣言”的文档在 2013 年发布，它聚焦于：如何在互联网场景中构建健壮可用的应用系统，如何在各种形式的外部访问（事件、关联调用、负载、错误异常）中保证系统的稳定性。Reaction 宣言可以说是对 12-Factor 的抽象总结。

人们在各自不同的业务领域从事软件构建工作，总结出了一套开发设计模式，它们看起来是如此相似，遵循这些模式的应用系统更加健壮、坚韧与灵活，更适应于现代要求。然而系统需求在近几年发生了戏剧性变化，这些模式也正随之发生改变。几年前，一个大型系统由数十台服务器构成，秒级的响应时间，小时级的离线维护以及 GB 级的数据量，到了如今，系统将被部署到任何地方，从终端用户的移动设备到上千颗多核处理器的云端集群服务器。用户期望的响应时间变为毫秒级，要求 100% 在线运行时间，而数据则开始以 PB 级衡量。之前的软件架构已不能再简单的适应于今天的需求。一套清晰的系统架构方法是必要的，这些必要的架构方面已被逐一地识别出来，宣言将其称之为“Reaction System”，它们是：响应（Responsive）、韧性（Resilient）、弹性（Elastic）、消息驱动（Message Driven）。

### 2.3.1 响应（Responsive）

系统应尽可能地及时地响应。响应是可用性与有效性的基石，除此之外，响应意味着问题被尽早发现、及时处理。响应系统聚焦在保证快速而一致的响应时间，建立可靠的时间上限，从而交付一致的服务质量。这种一致性反过来又简化了错误处理，建立终端用户信心，并鼓励进一步互动。

### 2.3.2 韧性（Resilient）

系统在遭遇故障（failure）时依然要保持响应。这不仅适用于高可用、业务关键类系



统，所有不具备韧性的系统在故障后都将无法提供响应。

故障特指非预期的阻碍持续服务的事件，一般来说其会阻碍当前以及随后的所有用户请求。这与错误（error）截然相反，一个错误是预期的、代码逻辑可处理的，例如在用户表单输入内容校验时引发的错误。较之故障让整个系统的处理能力下降，错误却并不是致命的。错误是常规操作的可预知部分，能被及时处理，错误之后系统能继续在相同能力水平下提供服务。故障的例子有硬件宕机、进程因重要资源耗尽而退出，程序缺陷引发内部瘫痪等。

韧性通过复制、封闭、隔离和委派来实现。故障存在于每一个组件中，组件之间的相互隔离可保证局部故障及其恢复不会影响到整体。节点的恢复被委派到另一个（外部）组件负责。高可用通过节点间的复制实现。客户端组件不参与异常处理中。

组件在不同位置同时运行被称之为复制。它可以是在不同线程或线程池、进程、网络节点或数据中心。复制提供伸缩性，进来的负载被分发到组件的多个实例（例如一般服务的负载均衡）。复制也提供健壮性，进来的负载被克隆到多个实例并行处理（例如大数据的并行计算）。

隔离（或封闭）能被定义为在时间和空间上的解耦。在时间上的解耦，意味着发送方与接收方能拥有相互独立的生命周期，它们并不需要为了相互通信而同时存在。通过在组件间引入异步的边界，采用消息传递的通信方式实现。在空间上的解耦（定义为位置透明）意味着发送方与接收方并不需要运行在相同的进程中，而是由运营部门或者程序本身所决定的最优效率的任何地方，并可以在应用生命周期中改变。

委派的目的在于将一个任务的执行保证交由另一个组件负责。这个组件可以执行其他工作，或随意地观察委托任务的进展情况，如果进一步动作（如故障处理或进展报告）需要的话。

### 2.3.3 弹性（Elastic）

系统在变化的工作负载下保持响应。Reactive Systems 能够感知外部输入请求速率的变化，通过减少与增加资源来做出反应。这意味着没有中心瓶颈、临界区的设计，对组件进行复制或分片，将输入请求分发到其上。Reactive Systems 支持预测，并可及时反应，伸缩算法建立在相关的实时性能数据指标上。他们在商业硬件和软件平台上实现性价比高的资源弹性。这种弹性意味着系统的吞吐量随着资源成比例的增加或减少而自动伸缩，从而适应外部请求的变化。

组件用于执行任务所依赖的任何东西都称为资源，它们必须按照组件需要而分配，包括 CPU、主存、存储、网络带宽、任务调度、时钟、输入输出以及类似于数据库、网络文件系统等外部服务。这些资源的可靠性以及可伸缩性必须考虑，因为缺少必要资源将影响

到组件某些功能的执行。

### 2.3.4 消息驱动（Message Driven）

**Reactive Systems** 依赖于异步消息传送，以在组件之间建立起边界，从而实现松耦合、隔离、位置透明，以及提供将错误委派给消息处理的手段。明确的消息传送机制通过创建、监控消息队列，并在必要时应用背压（back pressure）使负载处理、弹性伸缩、流量控制得以实现。位置透明消息作为一种通信手段，使得故障的管理可以在一个集群或一个主机上以相同语义结构工作。非阻塞（non blocking）通信允许接收者只在活动时消耗资源，导致系统开销更少。

背压是应对压力负载的一种反馈机制，它使系统能够优雅地响应负载，而不是突然的故障或者性能急剧下降。当压力负载达到一定程度即将无法应付时，他们会将情况反馈给上游系统，采取减少负载、重分发负载以及弹性扩容等方法规避异常。

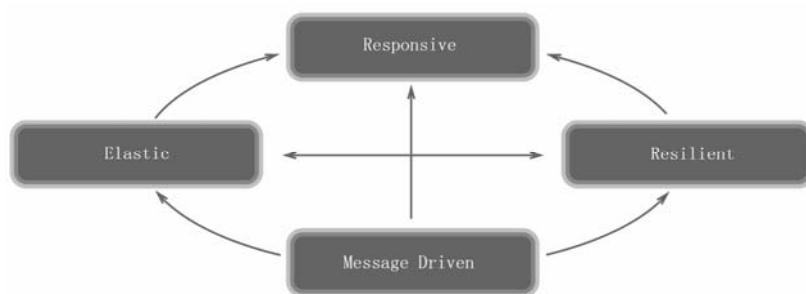


图 2-10 Reactive Systems



## 第二部分 基础原理

计算机系统核心资源是计算、存储、网络三要素，云计算将这三大资源进行池化，做到按需分配、弹性扩容、自助申请。云计算也好，分布式平台也好，从来就没有改变过这三大基础资源的原本形态。在这一部分中，我们将重拾计算机系统的本真，回到其原始形态。

### 3.1 图灵机与冯·诺伊曼模型

#### 1. 图灵机

阿兰·麦席森·图灵（Alan Mathison Turing, 1912-1954），是英国数学家、逻辑学家，被称为人工智能之父，他是计算机逻辑的奠基者，提出了有限状态自动机，也就是现在所说的图灵机。他在《机器能思考吗》论文中对图灵机进行了数学上的描述，并且进一步从哲学层面阐述其运转机制。他将该模型建立在人们进行计算过程的行为上，并将这些行为抽象到用于计算的机器模型中，从而真正地改变了世界。

图灵的基本思想是用机器来模拟人们用纸笔进行数学运算的过程，在所有问题的解决方法及序列确认的前提下，他将该过程看作下列两种简单的动作：

- （1）在纸上写上或擦除某个符号；
- （2）把注意力从纸的一个位置移动到另一个位置。

为了模拟人的这种运算过程，图灵构造出一台假想的机器，这就是图灵机。它可以读入一系列的 0 和 1，这些数字代表了解决某个问题所需要的步骤，按这个步骤走下去，就可以解决某个特定的问题。他并不将焦点放在问题的解决方法上，而是认为一切问题的解决方法由最小的逻辑单元组合而成，这个逻辑聚合被称为程序。如图 3-1 所示。

图灵机由以下几个部分组成。

（1）一条无限长的纸带 **TAPE**。纸带被划分为一个接一个小格子，每个格子上包含一个来自有限字母表的符号，字母表中有一个特殊的表示空白的符号。纸带上的格子从左到右依此被编号为 0、1、2、……纸带的右端可以无限伸展。

（2）一个读写头 **HEAD**。该读写头可以在纸带上左右移动，它能读出当前所指的格子上的符号，并能改变当前格子上的符号。

（3）一套控制规则 **TABLE**。它根据当前机器所处的状态，以及当前读写头所指的格子

上的符号来确定读写头下一步的动作，并改变状态寄存器的值，使机器进入一个新的状态。

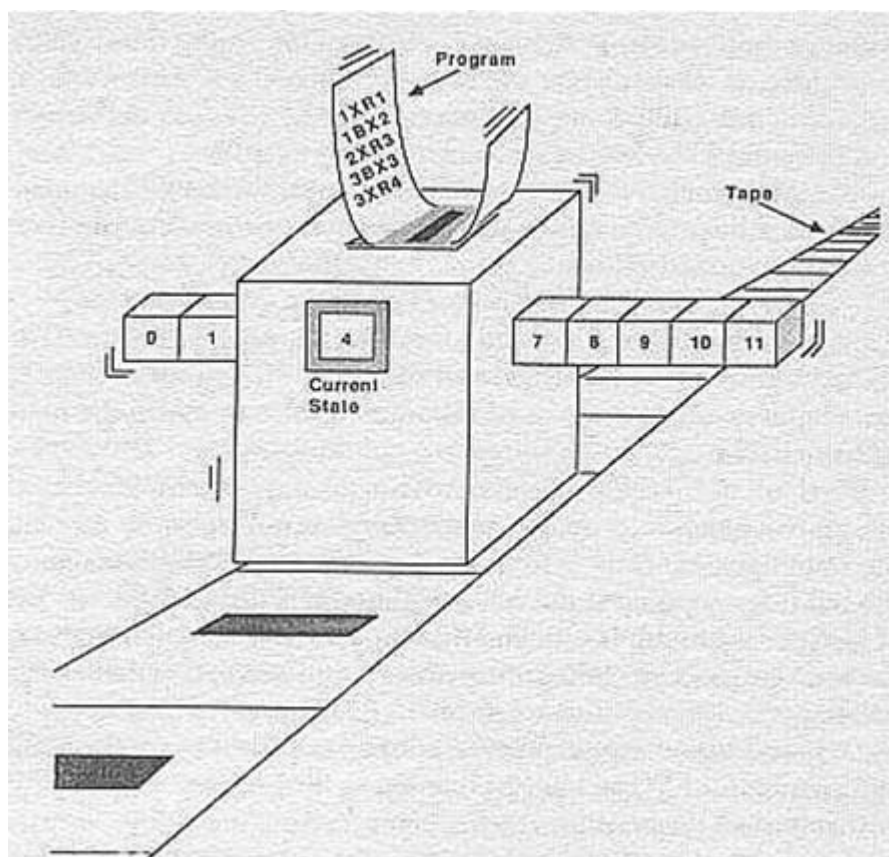


图 3-1 图灵机

(4) 一个状态寄存器。它用来保存图灵机当前所处的状态。图灵机的所有可能状态的数目是有限的，并且有一个特殊的状态，被称为停机状态。

图灵认为这台机器只用保留一些最简单的指令，一个复杂的工作被分解为这几个最简单的操作就可以实现了，进而模拟人类所能进行的任何计算过程，而其中的困难是如何确定最简单的指令集，什么样的指令集才是最少的并且管用的。

图灵模型是一个适用于通用计算的模型，该模型并不将计算功能绑定到一个特定的领域，而是期望其解决一切问题，能够让其保持通用性的奥秘在于在其中添加了一个额外的元素，即程序，程序是驱动计算机硬件对数据进行加工处理的指令集合。在这个模型中，输出数据依赖于输入与程序两方面的作用，对于相通的输入与程序，必然有一致输出。通用图灵机是对现代计算机的首次描述，只要提供合适的程序给该机器就能够适应任何领域的计算。

### 2. 冯·诺依曼模型

柏拉图认为世界上的各种事务总是有一个“理型”在后面，比如世界上一定有一个“马”的理型，才会有各种各样具体的马，它们都是那个理型产生的具体个体。我们人所处的物质世界就是由一个理型世界产生出来的，这个物质世界在不停地变化，然而那个理的世界是永恒不变的。

我们可以认为图灵机就是计算机系统的理型，而人类依据这个理型朝着实现的方向不断前进，推动这一切的有三类人：艺术家、科学家与疯子。

在图灵机的设想中只有数据是存放在存储器中的，而程序因为其独立性而很难改变。我们可以认为最早基于图灵机实现的计算机，其程序逻辑与硬件是绑定在一起的，服务于一个专用领域，例如一个文本处理计算机无法服务于图形处理领域，每次为新的领域服务就必须重新设计其程序逻辑，这样会在通用性上大打折扣。

在图灵机的基础上，冯·诺依曼在 1944~1945 年间指出，鉴于程序与数据在逻辑表示上具有一致性，因此程序也能存储在计算机存储器中。程序的可存储让程序在存储器中以一种特别形态的静态数据而存在，可以被轻易改变，这些静态数据代表一组指令集，而运算则是对这组指令集的顺序执行。冯·诺依曼提出数字计算机的数制应采用二进制，计算机应该按照程序顺序执行，因此对计算机的理型做了进一步改进。

基于冯·诺依曼建造的计算机分为 5 个部件：

- 存放数据和指令的存储部件；
- 对数据执行算术与和逻辑运算的算术逻辑部件；
- 把数据从外部世界转移到计算机内部的输入部件；
- 把结果从计算机内部转移到外部世界的输出部件；
- 协调各方的控制器部件。

冯·诺依曼模型如图 3-2 所示。

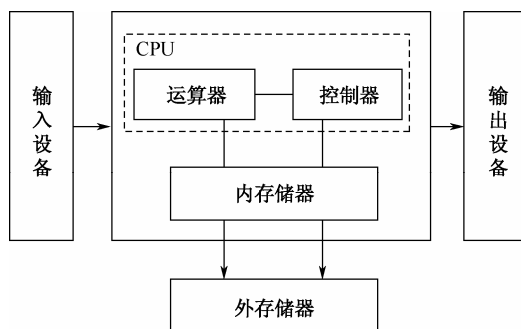


图 3-2 冯·诺依曼模型

1) 存储器

在计算机的处理过程中存储器是用来存储数据和程序的。在计算机系统中一切内容都是用二进制表示的，无论是数字、字符、图片，还是声音，我们将在下一节对此进行讨论。在每个存储单元中能够存放 1 或 0，这些位被组合成字节（8 位），字节被组合成字。内存是存储单元的集合，每个存储单元都有一个唯一的物理地址。计算机系统依据这个地址对内存进行操作，目前大多数计算机都是以字节为单元进行编址的。

内存的存储单元是从 0 开始连续编号的，例如，如果可编址性是 8 位，那么内存就有 256 个存储单元。内存存储单元的编号如图 3-3 所示。

地址	内容
00000000	11100011
00000001	10101100
⋮	⋮
⋮	⋮
⋮	⋮
11111101	10101100
11111110	10101110
11111111	10101110

图 3-3 内存存储单元的编号

2) 算术逻辑单元（ALU）

算术逻辑单元（Arithmetic/Logic Unit）是用来进行算术运算和逻辑运算的，算术运算即两个数的加法、减法、乘法和除法，逻辑运算，亦即与、或、非运算。

在图灵机中有专门计算中间状态的部件，而在大多数现代 ALU 中都有少量的特殊存储单元，被称为寄存器。寄存器能够容纳一个字节，用于存放在计算过程中下一次会立即用到的信息，也就是计算的中间状态信息。

3) 输入/输出部件

输入/输出部件是计算机和外部世界沟通的渠道，如果在计算过程中无法与外界交互，无法获取外界的值，或者无法将结果输出给外界，那么再强的计算能力也是无用的。

输入部件是外界数据和程序进入计算机的设备。古老的计算机系统输入可能是打孔的纸片，现代的输入设备包括终端键盘、鼠标、狼牙接入等。

输出部件使外界能够使用最终计算出的结果。最常用的输出设备是显示器、打印机。

#### 4) 控制器

控制器掌握着读取-执行周期，它是计算机中的总控制中心。在控制器中有两种寄存器。指令寄存器存放着正在执行的指令，程序计数器存放着下一条执行指令的地址。ALU 和控制器的协作非常紧密，所以它们常被当作一个部件，被称为 CPU（Central Process Unit，中央处理器）。

## 3.2 服务器的种类

沿着历史的脚步往前走，冯·诺依曼模型在计算机领域仍然占据着统治地位，自 20 世纪 90 年代并行处理器进入市场后，才在原有体系上做了些许修改。计算机发展到现在已无处不在，桌面 PC、企业服务器、移动手机等，它们已经深入到了人们生活的方方面面，在这里让我们关注一下数据中心的企业服务器。

服务器往往被用于运行企业或个人的关键业务，所以对其性能与可靠性方面的要求会远远高于个人 PC。一般来说，服务器的 CPU、内存、网络、存储都会使用企业级部件。例如相比桌面电脑常用的 Intel 酷睿系列 CPU，服务器的 CPU 往往会采用性能更稳定、强大的 Intel 至强（Xeon）系列或者 IBM 的 Power 系列。而内存也会采用带有诸如 ECC 校验等自恢复功能的高速企业级内存。为了满足特定的业务需求，不同种类的服务器又会在网络、存储、内存、显卡等方面进行强化。

按照 CPU 体系架构来区分，企业服务器主要分为以下两类。

非 x86 服务器：包括大型机、小型机和 UNIX 服务器，它们使用 RISC（精简指令集）或 EPIC 处理器，并且主要采用 UNIX 和其他专用操作系统。精简指令集的处理器主要有 IBM 的 POWER 和 PowerPC 处理器，SUN 与富士通合作研发的主要有 SPARC 处理器、EPIC 处理器，HP 与 Intel 合作研发的主要有安腾处理器等。这种服务器价格昂贵，体系封闭，但是稳定性好、性能强，主要用在金融、电信等大型企业的核心系统中。

x86 服务器：又叫作 CISC（复杂指令集）架构服务器，即通常所讲的 PC 服务器，它是基于 PC 机体系结构，使用 Intel 或其他兼容 x86 指令集的处理器芯片得到诸如 Linux、Windows 等操作系统的广泛支持，IBM 的 System x 系列服务器、HP 的 ProLiant 系列服务器等都属于 x86 服务器，较之大型机，它们价格便宜、兼容性好。虽然在稳定性方面相对逊色，但 x86 服务器可以满足绝大多数业务应用场景。

#### 1) CISC 型 CPU

CISC（Complex Instruction Set Computer，复杂指令集）指 Intel 生产的 x86（Intel CPU



的一种命名规范)系列 CPU 及其兼容 CPU (其他厂商如 AMD、VIA 等生产的 CPU), 基于 PC 机 (个人电脑) 的体系结构。这种 CPU 一般都是 32 位的结构, 可称为 IA-32 CPU (IA 是 Intel Architecture 的缩写, 即 Intel 架构)。CISC 型 CPU 目前主要有 Intel 的服务器 CPU 和 AMD 的服务器 CPU 两类。

## 2) RISC 型 CPU

RISC (Reduced Instruction Set Computing, 精简指令集) 是在 CISC (Complex Instruction Set Computer) 指令系统的基础上发展起来的, 相对于 CISC 型 CPU, RISC 型 CPU 不仅精简了指令系统, 还采用了一种超标量和超流水线的结构。架构在同等频率下, 采用 RISC 架构的 CPU 比采用 CISC 架构的 CPU 性能高很多, 这是由 CPU 的技术特征决定的。RISC 型 CPU 与 Intel、AMD 的 CPU 在软件和硬件上不兼容, 如图 3-4 所示为 IBM 在 2014 年上市的 Power ISA 2.07 CPU。

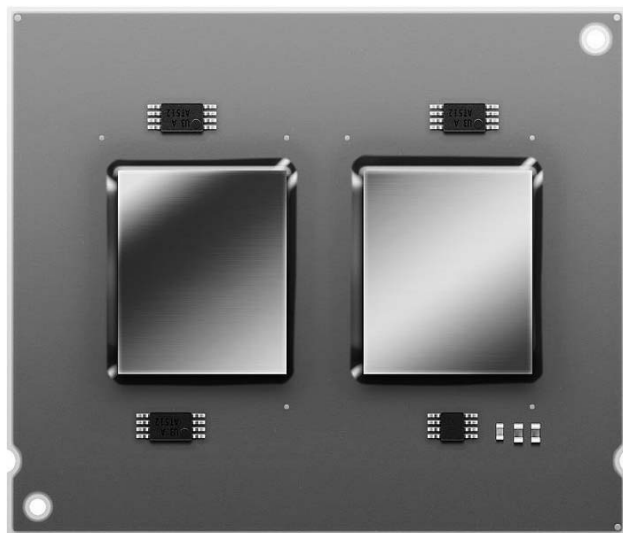


图 3-4 IBM Power ISA 2.07 CPU

根据不同的业务需求与应用场景, 可以将服务器架构分为通用式服务器、机架式服务器、刀片服务器和企业服务器。下面以 IBM System x 为例介绍这些架构的组成、特点与适用场景。

## 1) 直立式服务器 (塔式服务器)

直立式服务器为可独立放置于桌面或地面的服务器, 大多具有较多的扩充槽及硬盘空间, 不需要增加额外的设备, 插上电即可使用, 因此使用最为广泛。在外形上和立式桌面电脑很相似, 不过其主板的可扩展性更高。这种服务器一般使用入门级至强处理器, 也可以使用低成本的酷睿、奔腾、赛扬系列, 适用于中小企业的文件与打印、电子邮件与协作、防火墙、分布式分支机构等业务。如图 3-5 所示。



图 3-5 IBM X300M2 塔式服务器

## 2) 机架式服务器

机架式服务器在外形上有别于桌面计算机，更像交换机，其使用工业标准尺寸，可以安放在标准的 19 英寸机柜中。与塔式服务器相比机架式服务器更节约空间，在性能上也更强大。机架式服务器为可装机柜的服务器，主要作用是节省空间，机台高度以 1U 为单位，1U 约 44mm，因空间较局限，扩充性较受限制，例如 1U 的服务器大多只有 1~2 个 PCI 扩充槽。此外，散热性能成为十分重要的因素，此时，各家厂商的功力就在此展现了。缺点是需要有机柜等设备，多被服务器用量较大的企业使用。

这种服务器一般采用中端的至强系列处理器，适用于文件与打印、零售、网络基础架构、分布式应用程序、Web 服务、Web 2.0、虚拟化、光数据库、ERP 应用程序、小型 HPC 安装、ERP 部署和虚拟化。如图 3-6 所示。



图 3-6 戴尔服务器 PowerEdge R510

### 3) 刀片服务器

刀片服务器是比机架式服务器更节省空间的产品。主要结构为一大型主体机箱，内部可插上许多卡片，一张卡片即相当于一台服务器。当然，散热性也非常重要，各家厂商往往装上大型强力风扇来散热。这种服务器虽然较节省空间，但光是主体机箱部分就可能价格不菲，一般只有大型企业才会使用，如银行、电信、金融行业及互联网数据中心等。如图 3-7 所示。



图 3-7 浪潮英信 NX4120

## 3.3 一切都是二进制

冯·诺依曼模型是一套信息处理机模型，阐述的是机器内各部件的分工协作；企业服务器及它内部的中央处理器为我们提供了计算工具。本节我们先不急着向操作系统进军，而是从更基础的底层来思考，即人类真实世界中的事务，以及事务变化在计算机系统中是如何表示的。

要让机器可以与外界交流，必须将外部信息转化为机器可以读写的物理表现，可惜的是机器无法直接辨别数字、字符和颜色。在计算机发展初期，人们用最原始的穿孔纸带来记录、表示信息，信息的元数据状态被控制得足够简单，只有两种：有孔、无孔。不要期待机器识别更多，比如将孔做成三角形、正方形、圆形来表示更多状态，当时的工程技艺虽然能够做到不同形状的孔，但其误报率是让人无法接受的。在计算系统的盘古开天地时期，物理状态的表示就已定型，“to be or not to be”，就此两种而已。工程科技持续向前发展，一群 IBM 工程师和科学家研发出新的存储设备，磁存储很快便替代了纸带。直到现在，几乎所有信息的物理存储与表示采用的都是电磁方式，一般来说，0~2V 的电压电平是低电压，2~5V 内的电压电平是高电压，高低电压分别代表着 0 和 1。高低电平、南北磁极延续了最初两种状态的表示。物理上的表示法则让人们决定采用二进制来表示真实的世界。

计算机最初能够处理的都是数字和文本，但随着科学家对各类媒体信息进行标准定义，计算机已经可以处理各种各样的信息了。计算机可以存储、表示各种类型的数据，包括数字、文本、音频、图像和视频。

二进制的位只能是 0 或 1，一个位只能表示两种状态之一，例如，我们将水的温度分为冷和热两种，将旗帜的颜色分为红色和黄色，这只需要一位二进制数即可。但如果要表达更多的分类，那么一位二进制数就无法满足了。要表示多于两种的状态，则需要多个位。两个位可以表示四种状态，其可以构成 00、01、10 和 11，例如，我们将一面旗帜所组成的颜色分为红、蓝、白、灰。随着表示的状态越来越多，我们需要两个以上的位。3 位二进制可以表示 8 位， $n$  位二进制可以表示  $2^n$  种状态。我们需要将计算机所理解（读写）的二进制与人类所理解的计数系统分开，计算机的二进制仅仅表示其可以容纳多少状态。为了描述一个事物，根据我们需要多少状态，则选择一个满足状态数的二进制位长度。而人类的计数系统是将两个数进行加、减、乘、除运算后得出新的结果。

计算机能够处理的对象全部采用二进制表示，下面讲解数字的表示法。

### 3.3.1 整数表示法

对数字 1 我们用 001 表示，二进制共有 3 位， $2^3$  可表示 8 个数值，按人类常规思考方式在前面加上一个符号位，1 表示负数，0 表示正数，-1 则用 1001 来表示。符号表示法通过左边第一位表示数所属的分类（正数或是负数），剩余部分表示数的量值。符号表示法有一个问题，表示 0 的方式有两种：+0 和 -0；0000 和 1000，如果说我们并不在乎一个状态位的损失，那么实际上我们还有其他数值的表示法。

先来看如图 3-8 所示的数字表盘。

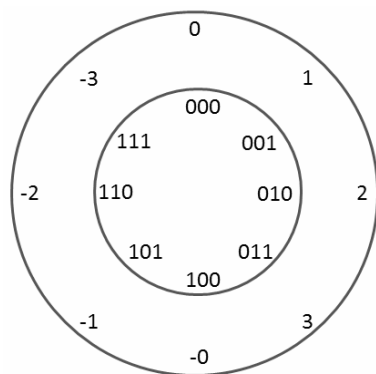


图 3-8 数字表盘

在一个 3 位二进制的数值盘中有两个 0 存在，在计算 1 减 3 时，实际上是对 1 (001) 和 -3 (111) 求和，求和的值是 1000，并不是我们期望的 -2 (110)。利用了人类习惯性思维的符号表示法在进行数值计算时却出现了问题，无法一步到位地计算出正确的结果，即正确的状态符 (110) 和其所表示的值 -2。

另外一种数值表示法被称为补码表示法，如图 3-9 所示的是 3 位二进制的补码表盘。

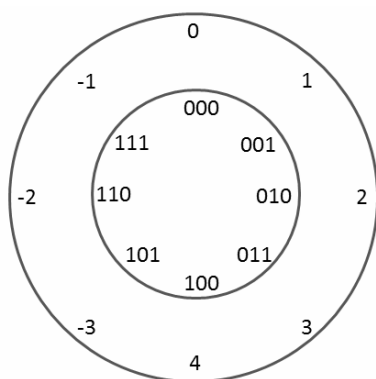


图 3-9 补码表盘

将 8 个状态位从圆心中轴线劈开，左右交替表示，1 表示 001，-1 表示 111，从圆形坐标上看，一个值的负数位于其中轴的对面。表示正值时，与符号表示法相同。表示负值时，先把正数值取反，即把所有的“1”转换为“0”，把所有的“0”转换为“1”，之后再加上 1。补码表示法有什么意义？我们再回过头看 1 减 3 的计算，在 1（001）和-3（101）之间求和，其值正好等于-2（110），也就是说用补码表示法来表示数值，在两个数值之间进行计算时并不需要经过复杂的步骤，可一步到位。

### 3.3.2 文本表示法

一个文本可以被分解成段落、句子、词、标点符号。我们可以将文本中的内容拆分成独立的字符，制订字符集标准，对每一个字符设置一个二进制状态码，比如英语有 26 个字母，但必须有区别地处理大写字母和小写字母，所以实际上有 52 个字母，另外还有数字（0~9），以及各种标点符号。我们可以预估所需要的状态码数量，选择合适的位长，定制一个字符集，即字符和表示它们的代码对应关系的清单，字符集是字符和表示它的二进制状态码的清单。

ASCII 码（American Standard Code for Information Interchange）是早期最有名的字符集编码，它是为英文通信而设计的，由 128 个字符组成，用一个字节来表示。表示的字符分别是：大小写字母、数字、标点符号、非打印字符及控制符。如表 3-1 所示。

ASCII 字符集的扩展版本提供了 256 个字符，虽然足够表示英语，但是无法满足国际需要。这种局限性导致了 Unicode 字符集的出现，Unicode 的创建者的目标是表示在世界上使用的所有语言中的所有字符。此外，它还表示了许多补充的专用字符，例如科学符号。Unicode 字符集使用 16 位表示每个字符，其可以表示 216 个字符，即 65000 多个字符，远远超过了 ASCII 的 256 个。为了保持一致，Unicode 字符集被设计为 ASCII 的超集。也就是说，Unicode 字符集中的前 256 个字符与扩展 ASCII 字符集中的完全一样，表示这些字符的代码也一样。

表 3-1 ASCII 字符集

高 四 位  <	
--	--

有了 Unicode 编码似乎所有的问题都解决了，事实并非如此。人们发现 Unicode 用于表示 ASCII 码的效率非常低，因为 Unicode 编码表示 ASCII 字符集时，要比 ASCII 编码多出一倍的空间，而且对于 ASCII 码来说高位的 0 对于它没有任何意义。于是伟大的设计师们又设计出了一种中间格式字符集，也叫作通用换码格式的字符集。UTF-8（8-bit Unicode Transformation Format）就是其中一种。

UTF-8 是一种针对 Unicode 的可变长度的字符编码，也是一种前缀码。它可以用来表示 Unicode 标准中的任何字符，且其编码中的第一个字节仍与 ASCII 兼容，这使得原来处理 ASCII 字符的软件无须或只做少部分修改，即可继续使用。因此，它逐渐成为电子邮件、网

页及其他存储或发送文字的应用中，优先采用的编码。

### 3.3.3 音频信息表示法

声音是一种压力波，当演奏乐器、拍打一扇门或者敲击桌面时，声音的振动会引起空气分子有节奏地振动，使周围的空气产生疏密变化，形成疏密相间的纵波，这就产生了声波，这种现象会一直延续到振动消失为止。当一系列的空气压缩震动我们的耳膜时，我们的大脑接收到一个信号，于是我们感觉到了声音。

作为波的一种，声音的频率和振幅就成了描述波的重要属性，频率的大小与我们通常所说的音高对应，而振幅影响声音的大小。声音可以被分解为不同频率、不同强度的正弦波的叠加。这种变换（或分解）的过程被称为傅立叶变换（Fourier Transform）。

因此，一般的声音总是包含一定的频率范围。人耳可以听到的声音的频率范围为 20~20000Hz。高于这个范围的波动被称为超声波，而低于这一范围的波动被称为次声波。狗和蝙蝠等动物可以听到高达 160000Hz 的声音。鲸和大象则可以发出频率为 15~35Hz 的声音。

音调、响度、音色是声音的三个主要特征，人们根据它们来区分声音。

- 响度：人们主观上感觉到的声音的大小（俗称音量）由振幅（Amplitude）和人离声源的距离决定，振幅越大，人和声源的距离越小，响度越大，单位为分贝（dB）。
- 音调：声音的高低（高音、低音）由“频率”（Frequency）决定，频率越高音调越高（频率的单位为 Hz）。低音端的声音或更高的声音如同细弦声。频率是每秒经过一给定点的声波数量，它的测量单位为 Hz，是以海因里希·鲁道夫·赫兹的名字命名的。此人设置了一张桌子，演示频率是如何与每秒的周期相关的。1kHz 或 1000Hz 表示每秒经过一给定点的声波有 1000 个周期，1MHz 就是每秒钟有 1 000 000 个周期，等等。
- 音色又称音品，波形决定了声音的音色。声音因不同物体材料的特性而具有不同的特性。音色本身是一种抽象的东西，但波形是这个抽象直观的表现。音色不同，波形不同。典型的音色波形有方波、锯齿波、正弦波、脉冲波等。不同的音色通过波形则完全可以分辨。

一个立体声系统通过把电信号发送到一个扬声器来制造声音。这种信号是声波的模拟表示法。信号中的电压按声波的正比例变化。扬声器接收到信号后，将引起膜震动，引发空气震动，从而引起耳膜震动。

要在计算机上表示音频信息，则必须数字化声波，也就是将模拟信号转换为数字信号，把连续的声音分割成离散的、便于管理的片段。其方法是采集表示声波的电信号，用

一系列离散的数值表示它。模拟信号是随电压连续变化的。要数字化这种信号，则需要周期性地测量信号的电压，记录电压值，进行采样，最后得到用不同电压电平表示的一系列不连续的数字信号。采样的频率越高，人耳听到的声音就越接近于真声。

## 3.4 操作系统——计算机系统的指挥官

确定了信息的二进制表示法（文字、图片、声音），也拥有了计算机系统的模型（冯·诺依曼模型），程序员依照计算机指令集来编写程序并在计算机系统中运行。那么问题来了，不管你是一个运维人员还是开发人员，你会发现我们在编写程序的过程中很少直接与 CPU 的指令集打交道，也就是常说的机器码，同时也不会关注真实的物理内存地址。在这个计算机系统模型上，有一个强大的资源管理者，它掩盖了底层物理设备的复杂性、多样性，通过一组标准接口让我们轻松编写自己的程序并使其稳定运行，它就是操作系统。

### 3.4.1 操作系统解决的问题

计算的本真是输入、输出与计算，操作系统本身也是一个程序，也有自己的逻辑，它的出现是为了解决以下 4 个问题的。

#### 1) 掩盖底层硬件设备的复杂性

我们完全可以绕过操作系统直接与硬件设备打交道，在 BIOS 加电启动的过程中，它会在默认加载设备的固定地址读取启动程序，如果没有操作系统，那么我们必须面对层出不穷的硬件设备，我们要初始化所有可以访问的设备，按照设备厂商提供的指令集手册向设备占据的端口发送原始的操作指令，从而控制、使用设备。

有了操作系统对硬件设备的统一管理，其向上对应用程序提供统一的抽象接口，open 系统调用可以打开不同类别的设备文件，向下要求设备商依据抽象接口编写自己的驱动程序。

#### 2) 硬件资源调度与分配

在计算机系统中，一般需要“同时”运行多个程序，它们担当不同的角色，完成不同的任务。我们必须将资源（CPU、内存、IO）分配给这些任务，它们才能活下去。然而真实的物理资源是有限的，单核 CPU 资源无法同时提供给多个任务使用，于是操作系统又发挥了其神奇功能。通过资源调度、上下文切换让不同的任务分享 CPU 资源，通过虚拟地址空间让每一个任务都看到了一个连续的、充足的物理内存。

#### 3) 提供统一的程序运行环境

程序的运行过程有很大一部分需要与外部设备交互，存储数据需要内存空间，读写文件需要访问磁盘，对于唯一的访问“地址”，我们不可能在程序运行时才依据当前情况去更



新访问资源的物理地址。操作系统为每一个程序的运行环境提供了一个与现实环境无关的空间，每一个任务都拥有自己的虚拟机地址，操作系统会将它映射到物理地址。操作系统还提供了统一的系统调用（System Call），通过抽象的接口访问底层设备。

#### 4) 在多任务下的系统安全性

最后，在一个多任务的计算机系统中如何保证任务间的隔离性，以及如何时保证一个任务不受另一个任务的影响也是操作系统需要考虑的。更进一步，为了不破坏底层物理资源的平衡性，操作系统通过 CPU 的寄出器设置了特权级别，对自身也进行了有效保护。

### 3.4.2 企业级操作系统

企业级操作系统领域主要分为三大阵营：UNIX、Linux 与 Windows。Linux 的诞生与 UNIX 有渊源。

#### 1. UNIX

UNIX 于 1969 年由 AT&T Bell Labs 发明，是一种健壮、灵活且对开发人员友好的计算环境。UNIX 在很长一段时期内以老大哥的身份占据着企业级操作系统市场。它与小型机服务器结合，采用专有的精简指令集 CPU 来支撑企业的关键应用。随着互联网的兴起，应用架构与数据架构发生了很大的变化，对 UNIX 操作系统的依赖越来越小，只有一些金融、电信等行业的关键业务系统运行在 UNIX 上。

##### 1) IBM AIX UNIX

AIX 是 Advanced Interactive eXecutive 的缩写，于 1986 年 1 月推出，是 IBM 发布的 UNIX 操作系统。最初，AIX 运行在 IBM RT/PC (AIX/RT) 上。自从 1989 年以来，AIX 成为 RS/6000 系列工作站和服务器的操作系统。在 AIX 的开发过程中，IBM 和 INTERACTIVE Systems Corporation (同 IBM 签约) 将 4.2 BSD 与 4.3 BSD 的一些特性加入了 AIX 中。于 2001 年推出的 AIX 5L 是 AIX 发展史上的重要版本，基于标准的开放操作系统，符合 The Open Group 的 Single UNIX Specification Version 3 的要求。它为以各种可伸缩性并发运行的 32 位和 64 位应用程序提供完全集成支持。AIX 5L 支持 IBM eServer p5、IBM eServer pSeries、IBM eServer i5 和 IBM RS/6000 服务器产品系列，以及 IntelliStation POWER 和 RS/6000 工作站。2007 年 11 月，伴随着新一代 Power CPU 即 Power 6 的诞生，IBM 也推出了最新的 AIX 6 操作系统。它在很多方面都进行了革命性的改变和创新，如虚拟化技术、安全架构、开发环境，以及持续可用性等方面。

在专有操作系统上，基础的应用环境包括包管理、编译器等都有着很大的差异。

- IBM SDKs for AIX、Java 2 Technology Edition 可以提供部署企业应用程序所需的卓越性能和稳定性。

- XL C Enterprise Edition V8.0 for AIX 是一款非常先进的基于标准的 C 编译器，专门用于运行 AIX 5L V5.1、V5.2 和 V5.3 操作系统的计算机。
- IBM XL Fortran Enterprise Edition for AIX 是一款适合科学家、工程师和应用程序编程人员使用的高性能编译器。
- IBM COBOL Set for AIX 可以将 IBM COBOL 技术应用到 AIX 上。这个强大的多用途 COBOL 应用程序开发环境提供了用于创建任务关键型客户端 / 服务器应用程序的编译器和支持工具。
- IBM 伴随 PowerVM 虚拟化技术推出的 Lx86 功能可以让你在 x86 上开发的 Linux 应用，无须任何迁移即可顺利地运行在 AIX 平台上。

### 2) HP-UX

HP-UX，是 Hewlett Packard UNIX 的缩写，是惠普以 System V 为基础所研发成的类 UNIX 操作系统，它的第一版于 1984 年发布，基于 System V release 3，只能在 RISC -PA-RISC HP 9000 平台上运行。Version 9 引入了基于字符的图形用户界面（GUI）SAM，让用户可以管理系统而不需要使用命令行。Version 10 于 1995 年引入，它修改了系统文件和目录结构的布局，在许多方面与 AT&T SVR4 更相似了。Version 11 于 1997 年被引入，这是第一个支持 64 位寻址的版本。11i 于 2000 年被发布，它引入了操作环境，也就是用于特定 IT 用途的分层的应用程序组。2001 年，Version 11.20 引入了对 Itanium 系统的支持。有意思的是，HP-UX 是第一种使用 Access Control Lists（ACL）管理文件权限的 UNIX。它还首先引入了对 Logical Volume Manager 的内置支持。当前最新的版本是 HP-UX 11i v3 Update 14。

### 3) Solaris UNIX

Solaris 原先是 Sun Microsystems 公司开发的类 UNIX 操作系统，后来 Sun 公司被 Oracle 并购，其改名为 Oracle Solaris。Sun 的操作系统最初叫作 SunOS，以 SunOS 5.0 开始，SUN 的操作系统开发开始转向 System V 4，并且有了新的名字，叫作 Solaris 2.0；Solaris 2.6 以后，SUN 删除了版本号中的“2”，因此，SunOS 5.10 就叫作 Solaris 10。Solaris 的早期版本后来又被重命名为 Solaris 1.x，所以“SunOS”这个词被用作专指 Solaris 操作系统的内核，因此 Solaris 被认为由 SunOS 图形化的桌面计算环境及其网络增强部分组成。

目前最新版为 Solaris 11。2005 年 6 月 14 日，Sun 公司将正在开发中的 Solaris 11 的源代码以 CDDL 许可开放，这一开放版本就是 OpenSolaris。2010 年 8 月 23 日，OpenSolaris 项目被 Oracle 中止。2011 年 11 月 9 日，Solaris 11 被发布。

Solaris 运行在两个平台上：Intel x86 及 SPARC / UltraSPARC。后者是 Sun Microsystems 公司使用的处理器。因此，Solaris 在 SPARC 上拥有强大的处理能力和硬件支持，同时 Intel x86 上的性能也正在得到改善。对这两个平台，Solaris 屏蔽了底层平台的差异，为用户提供尽可能一样的使用体验。Solaris 应该是目前几大 UNIX 平台在用户体验上做得最好的操作

系统，其文档支持也相当丰富。如图 3-10 所示。



图 3-10 Oracle Solaris

## 2. Linux

Linux 操作系统在互联网时代下发展迅猛，紧跟 IT 时代的步伐，其开发性、自由性适应了云计算、大数据下的需求。随着应用架构的变化，很多持久化性的数据应用需求也逐渐开始迁移到 x86 体系结构下的 Linux 服务器上。

### 1) RedHat Enterprise Linux

RedHat 创立于 1994 年，其在 Linux 行业享有盛誉。RedHat Linux 操作系统位居企业应用相关的 Linux 销售榜的榜首。RedHat 也是迄今为止最成功的 Linux 企业。它从第一天起就将关注企业性能、可用性和安全性需求铭记于心，在企业配置方面一直占据领先地位，从来没有被取代过。如今 RedHat 不仅仅专注于企业级 Linux，其在云计算、虚拟化、中间件等领域也都有所建树。RedHat 的 Linux 分为两个系列，其中一个是由 RedHat 公司提供收费技术支持和更新的 RedHat Enterprise Linux 系列；另一个是由社区开发的免费的 Fedora 系列。RedHat 的发展战略是每 3 年发布一个新版本的企业级 Linux 操作系统，并且每 6 个月发布一次更新。目前最新的发行版是基于 Linux kernel 3.11 的 Red Hat Enterprise Linux 7。如图 3-11 所示。



图 3-11 RedHat Linux

### 2) CentOS

作为于 2003 年年底才正式诞生的发行版，CentOS 是一个旨在对 RedHat Enterprise Linux（简称 RHEL）源代码进行重建，从而使其转化为可安装 Linux 版本的项目。该项目同时希望能够为所包含的软件包提供定期安全更新。如果选择更为直白的表达方式，那么 CentOS 其实就是 RedHat 的一套克隆版本。有人开玩笑地说，两个发行版之间唯一的技术性差异仅仅在于商标——CentOS 将原本的 RedHat 商标换成了自己的商标。如图 3-12 所示。



图 3-12 CentOS Linux

### 3) SUSE Linux

SUSE Linux 的起源可以追溯到 1992 年，当时有 4 位来自德国的 Linux 爱好者——Roland Dyroff、Thomas Fehr、Hubert Mantel 与 Burchard Steinbild——以 SUSE（即软件与系统发展）Linux 为名字创立了该项目。SuSE Linux 于 2003 年年底被 Novell 收购，于 2005 年 10 月推出了其自由软件版本 openSUSE。时至今日，openSUSE 已经有了对其赞赏有加的大量忠诚追随者。之所以能够获得用户的广泛认可，是因为 openSUSE 拥有令人愉悦而且精雕细琢的桌面环境（KDE 与 GNOME）、出色的系统管理方案（YaST），以及购买盒装版本的用户提供的适用于任何版本的优秀的书面说明文档。

### 4) Debian Linux

Debian Linux 最初公布于 1993 年。作为其创始者，Ian Murdock 最初的设想是组织成百上千位志愿开发者，利用他们的业余时间打造出一套完整的非商业项目。不到 10 年，它已经成为规模最大的 Linux 发行版，甚至有可能成为有史以来规模最大的协作软件项目。通过下面这些数字，大家可能会对 Debian GNU/Linux 的成功拥有更为具体的概念。目前参与过项目开发的志愿开发人员已经超过 1000 名，其软件库中包含 20000 多个软件包（被编译至 11 种处理器架构中），而且累计推出的基于 Debian 的发行版及 LIVE CD 版已经超过 120 种。Debian 主要分三个版本：稳定版本（Stable）、测试版本（Testing）、不稳定版本（Unstable）。目前的稳定版本为 Debian Jessie。Debian 是一款顶级 Linux 软件，很多产品使用 Debian 作为它们的基础软件，比如 Ubuntu、Mint。如图 3-13 所示。



图 3-13 Debian Linux

### 5) Ubuntu

Ubuntu 项目的发起人 Mark Shuttleworth 是一位极富个人魅力的南非富豪。他曾经参与过 Debian 的开发工作，同时是世界上第 2 位报名参加太空旅行的人。Ubuntu 的首个版本——Ubuntu 4.10 被发布于 2004 年 10 月 20 日，它以 Debian 为开发蓝本，可以说是 Debian 下的

一个分支。与 Debian 稳健的升级策略不同，Ubuntu 每 6 个月便会发布一个新版本，以便人们实时地获取和使用新软件。Ubuntu 的兴起在于对个人桌面用户体验的良好改善，后来开始同时提供针对企业应用的服务器版本。由于发源于个人桌面，Ubuntu 也许是最容易上手且易于使用的 Linux 操作系统，内置虚拟化和云连接等特性使得它成为所有 Linux 软件中的“万能”产品。如图 3-14 所示。



图 3-14 Ubuntu Linux

服务节点多、技术实力强的企业一般会选择 Ubuntu、CentOS、Fedora 等作为其企业级操作系统，甚至有的会在这些发行版本上进行源代码修改更新，并发布解决特定问题的企业内部版本。而对于中小型企业而言，它们更加关注的是稳定性及服务支持，在这种情况下，选择 RedHat、SUSE 是比较稳妥的做法。本书中的集群组件主要运行在 Ubuntu 上。

### 3.4.3 服务器虚拟化

虚拟化指将独立的资源放到一个大池中，之后再细粒度地进行资源分配。在单个服务器硬件平台上运行多个虚拟机（VM）的能力在如今的 IT 基础架构中实现了成本、系统管理和灵活性等方面的优势。在单个硬件平台上托管多个虚拟机，可减少硬件开支并最大限度地降低基础架构成本，比如能耗和制冷成本。虚拟化还提供了如今面向服务的高可用性 IT 操作中所需的操作灵活性，支持将正在运行的虚拟机从一个物理主机迁移到另一个主机，以满足对硬件或物理场所的需要，或者通过负载平衡最大限度地提高性能，或者应对日益增长的处理器和内存需求。

最早引入虚拟化技术的是大型机，在它们的专属操作系统上有着各自的虚拟化方式。随着 x86 虚拟化技术的发展，服务器虚拟化技术才得以快速发展与推广。虚拟操作系统通过虚拟机管理器（Virtual Machine Monitor, VMM）来访问实际的物理资源，按 VMM 的实现结构，VMM 可以分为以下三类。

#### 1. Hypervisor 模式

在 Hypervisor 模式中，VMM 首先是一个完备的操作系统，是为虚拟化而设计的，还具备虚拟化功能。从物理资源上看，所有的物理资源都归 VMM 所有，VMM 承担着管理物理资源的责任。其次，VMM 需要向上提供虚拟机以用于运行客户机操作系统，负责虚拟环境的创建和管理。VMware ESX Server 就是基于 Hypervisor 架构的。Linux KVM 是基于 GPL 授权的开源虚拟机软件，于 2007 年 2 月开始成为 Linux 内核的一部分，其项目发起人和维护人认为 KVM 是 Hypervisor 模型。

### 2. 宿主模式

在宿主模型中，物理资源是由宿主机操作系统管理的。宿主机操作系统是传统操作系统，本身不具备虚拟化功能，实际的操作系统由 VMM 来提供。VMM 通常是宿主机操作系统独立的内核模块，通过调用宿主机操作系统的服务来获得资源，实现处理器、内存和 I/O 设备的虚拟化。VMware 有多个虚拟化技术版本，其中 VMware Server 采用宿主模型，宿主机操作系统可以是 Windows 或者 Linux。

### 3. 混合模式

混合模型是两种模式的汇合体。VMM 依然位于最底层，拥有所有的物理资源。VMM 会将大部分 I/O 设备的控制权，交给一个运行在特权虚拟机中的特权操作系统。VMM 的虚拟化功能也被分担，处理器和内存的虚拟化依然由 VMM 来完成，而 I/O 的虚拟化则由 VMM 和特权操作系统共同合作完成。Xen 是一款基于 GPL 授权的开源虚拟机软件，起源于英国剑桥大学，属于混合模型，基于 Xen 的虚拟化产品有 Citrix、RedHat、Novell 等。

在商业产品上，VMware 是服务器虚拟化占比最高的厂商，其建立于 1998 年，于 2004 年被 EMC 收购控股。VMware 的虚拟化产品线非常丰富，对于 IDC 数据中心来说，我们可以将其划分为两种主要产品：VMware ESX Server 和 VMware Server。VMware ESX Server 是 VMware 在 IDC 的旗舰产品，属于 Hypervisor 模型，有着“近硬件”层级上的运行效率，它能使 x86 的利用效率提高 60%~80%。VMware Server 属于宿主模式，依赖于宿主环境的基本操作系统，在运行时会对服务器产生附加的开销。

在开源产品上，Linux KVM (Kernel-based Virtual Machine) 已经成为服务器虚拟化的主流，它是 x86 架构且硬件支持虚拟化技术（如 Intel VT 或 AMD-V）的 Linux 全虚拟化解决方案。KVM 虚拟化使用 Linux kernel 作为它的 VMM，在 Linux kernel 2.6.30 版本中已将对 KVM 的支持作为默认模块部分发布。使用 Linux 内核作为 VMM 成为对 KVM 的虚拟化模式定义的最大分歧点，很多人认为 KVM 并不属于 Hypervisor 模型，因为（在默认情况下）Linux 内核并不符合 Hypervisor 模型关于 VMM 的传统定义，即一个小操作系统，仅提供 VMM 运行所需的功能和驱动程序。但 KVM 的项目发起人却认为“小”只是一个相对的词汇，只要 VMM 功能在操作系统内核空间即可认为满足 Hypervisor 模型要求。

在企业级服务器虚拟化的选择上，追求稳定性与服务支持的企业会采用 VMware 的解决方案，技术实力强、有专人维护的公司则会采用 KVM 来降低企业成本，培植内部技术能力。

虚拟化技术的变化太快，而应用需求也在不断变化，企业应用架构的变化，将非核心、无状态化的服务拆解出来，这时之前保证资源隔离与安全性的虚拟机 guest OS 却变成了资源消耗大户。LXC (Linux Containers) 这种轻量级的虚拟化技术正适应了这种需求，不需要提供指令解释机制，以及全虚拟化的其他复杂性，对它而言，guest OS 所需要的一切在

宿主主机上都是共用的。LXC 不能像 VMM 虚拟化一样支持多种操作系统，却通过消减 guest OS 这一层资源消耗赢得了它的份额。

## 3.5 进程——资源聚合的抽象体

进程是操作系统里最成功的抽象体，它并不是真实存在的物理对象，而是一个程序的运行态，它将一个任务所需要的内存、计算、存储等资源聚合在了一起，并将其命名为 Process。

一个进程可以由一个或多个线程（Thread）构成，在最初的操作系统中，一个具体的任务需要并发、协同完成，相对于多进程独享各自的内存空间，线程的共享地址空间的处理效率要更高，而其在并发时的隔离性控制更加复杂。线程的另一个功能是在程序执行流中一个线程遇到 I/O 阻塞等待时，另一个线程可以继续执行任务，从而避免整个工作流的堵塞。

进程是程序的运行状态，那么程序是什么？程序是包含了二进制格式的代码和数据，其由源代码编译而成。编译是一个复杂的过程，编译器将高级语言转换为汇编语言，汇编器将汇编语言转换为机器语言的目标对象，链接器将多个目标文件合并成一个文件，加载器读取这些目标文件并将它们加载到内存中。

如果我们把进程看成分布式系统中的最小执行单元，那么为了能在任何地理位置执行，我们需要提前做好程序文件，并通过网络发送到计算单元上执行。

操作系统是封装硬件复杂性、提供标准接口的程序，而应用开发人员专注于业务逻辑领域，他们并不会（也不想）花太多时间与操作系统打交道，他们不会关注如何编写一个 Web 服务器、一个缓存中间件等，这时，在操作系统与应用程序之间又诞生了一类程序，我们常常称之为中间件。软件开发的核心思想是分层，各司其职、各尽所能。通过统一标准的接口进行约定。中间件的类型有很多，它在操作系统之上，且在应用逻辑之下，包括了服务处理、事务、消息队列等功能。

### 3.5.1 计算单元的构建

一个完整的计算任务运行起来依赖于本地操作系统、中间件及应用程序，它通过存储输入输出文件，以及网络与外界进行交互。为了实现大规模计算能力的快速伸缩，我们需要将计算任务所依赖的二进制代码在整个分布式平台中进行复制。

#### 1. 通过配置管理工具构建

纯人工方式地构建计算单元包括下面的步骤：

- 安装符合要求的操作系统；
- 下载相关软件并安装（源代码编译、二进制复制）；
- 编辑配置文件（例如 xml、txt，其中保存着环境信息）；
- 上传应用逻辑单元；
- 设置环境变量；
- 执行程序启动命令。

我们可以统一底层的操作系统，之后在其上通过一个轻量级的代理程序自动执行其上的所有活动，在这个执行过程中会与外部的中央管理程序进行交互。Chef、Puppet、Saltstack 是流行的配置管理工具。下面是 Chef 安装 JDK8 的一个例子：

```
# install jdk8
java_ark "jdk" do
  url 'http://download.oracle.com/otn-pub/java/jdk/8-b132/jdk-8-linux-x64.bin'
  checksum
'a8603fa62045ce2164b26f7c04859cd548ffe0e33bfc979d9fa73df42e3b3365'
  app_home '/usr/local/java/default'
  bin_cmds [ "java", "javac" ]
  action :install
end
```

通过配置管理工具构建计算单元的缺点是其还是依赖于一个标准的操作系统，另外用户要掌握配置管理的 DSL 语言，还需要一定学习曲线，在 PaaS 平台上向非运维用户提供配置管理工具可能并不合适。

### 2. 通过镜像构建

在 IaaS 上可以提供专门的镜像服务，这些镜像包括了用户的应用文件、中间件、操作系统及配置文件，我们可以利用它来生成一个最小的计算单元。虚拟机镜像的格式一般有 OVF、RAW、ISO 等。通过虚拟机镜像进行计算能力的劣势是：启动速度太慢，需从 OS 层开始启动；磁盘空间占用太多，每一个计算单元都独立占用自己的磁盘空间，其中放置了重复的操作系统、中间件程序文件。

### 3. 通过分离程序与进程构建

对于分布式平台上的计算逻辑的构建，我们并不想重复地复制操作系统乃至中间件的程序文件，这样会导致大量的存储空间被浪费，并不是每一个应用程序都需要一个操作系统，这样的结果是大量的计算资源也同样浪费在了操作系统上。

由运维人员直接管理的服务器，会在一个物理机器上安装一个操作系统，对不同的中间件进行分析：启动中间件进程依赖的执行文件是什么；它的输入及输出如何设置；如何通过设置环境变量、配置文件在一个 OS 上启动多个实例。这样的优势是所有资源得到了最



大限度的复用，启动速率要明显高于从 OS 级别开始的启动速率。这种方式的难度是运维人员必须非常熟悉中间件，能够将进程与程序干净地剥离出来，两个独立的进程之间没有任何影响。另外，进程间的安全控制是一个问题，不同的用户拥有不同的进程，却要使用相同的执行文件；在一个操作系统上如何有效地限制一个进程的使用资源，有效地隔离进程之间的影响的管理成本也很高。在中间件种类单一、标准可控、用户资源归属简单的情况下很适合用这种方式。

#### 4. 通过容器构建

容器可以理解为轻量级的虚拟机，它具有虚拟机的资源隔离性，但同时它又如同简单的进程一样在消耗最少的资源的情况下准备好了一个独立的运行空间。通过容器构建可很好地解决“分离程序与进程构建”的难点，运维人员无须再熟知其所要启动的中间件。Docker 是容器中的杰出代表，在后面的章节中我们将专门介绍。

### 3.5.2 计算请求的拆解

计算逻辑的构建是为了在任意节点都能够执行相同的逻辑、快速地复制、扩充计算能力。在计算的源头上，我们必须有办法将请求拆分、分发到计算节点。要想拆解计算，则需要对计算类型进行归类。

#### 1. Service 拆解

服务型计算主要指 Web 服务器在一定时间内响应用户的请求，返回计算结果。每一个请求的计算量并不大，但请求的数量随着用户量、业务量的增加而增加。

Service 类型的源头由大量重复的单一请求构成，任务拆解方式是从前端依据请求属性进行任务分发。在请求的数据传输路径上，有两个位置可以放置任务分发器。

第一个位置是 DNS 域名解析，Service 的地址是以域名的方式提供给用户进行访问的，在正常的请求过程中首先向 DNS 服务器询问目标域名的 IP 地址，我们可以放置自己的 SmartDNS，其对每一个域名配置多个服务 IP。每一个 IP 是计算逻辑相同的节点，计算能力通过加入更多的计算节点扩容。不同于一般的 DNS，SmartDNS 可以依据请求的相关属性（源 IP）进行任务分发，另外它还可以对后端服务进行健康检查，但节点在不可用时会被从集群动态中移除，以提升服务的可用性。DNS 的任务分发是轻量级的，因为随后的客户端、服务端通信不再经过它。

相对于 DNS 的轻量级任务分发，在客户端与服务端之间放置一个负载均衡设备是常用的方法，商用的设备有 F5；开源的软件有 LVS、Haproxy 等。不同于 DNS，负载均衡设备客户端与服务端的所有数据流都将经过它，虽然它不会有太多的计算逻辑，但由于请求都需经过它，承前启后及数据交互要建立大量连接，所以在海量请求的情况下，负载均衡设备本身会成为性能瓶颈。

## 2. Batch 拆解

对于 Batch 批处理类型的计算，每一次任务要对整个数据集进行计算；相对于大数据而言，其真正的处理瓶颈在于计算能力的不足，直接原因是数据的不可拆分导致计算的不可并发执行。Batch 批处理类型通常叫作离线任务，为了提升计算能力，我们会人为地对数据进行拆分，从而可以并发处理，提升计算能力。

MapReduce 是一种通用的大数据处理计算模型，已成为了这个领域的标准。其背后的原理是拆分大数据，复制计算逻辑，归并处理结果。我们往往关注的是数据，但反过来看，MapReduce 将 Batch 批处理类型的计算进行了拆分，从而可以扩展其能力。Hadoop 是 MapReduce 的开源实现，在分布式存储上采用 HDFS 来分割数据，而 MapReduce 模型负责计算逻辑的复制及结果的归并。MapReduce 的思想很简单，但要是这个模型接口下直接编写程序却有一定难度，软件世界的哲学永远是在复杂的问题上加上一层，让问题变得简单。例如在 Hadoop 中推出的 Hive 数据仓库工具，其在 MapReduce 之上，学习成本低，可以通过类 SQL 语句快速实现简单的 MapReduce 统计。如图 3-15 所示。

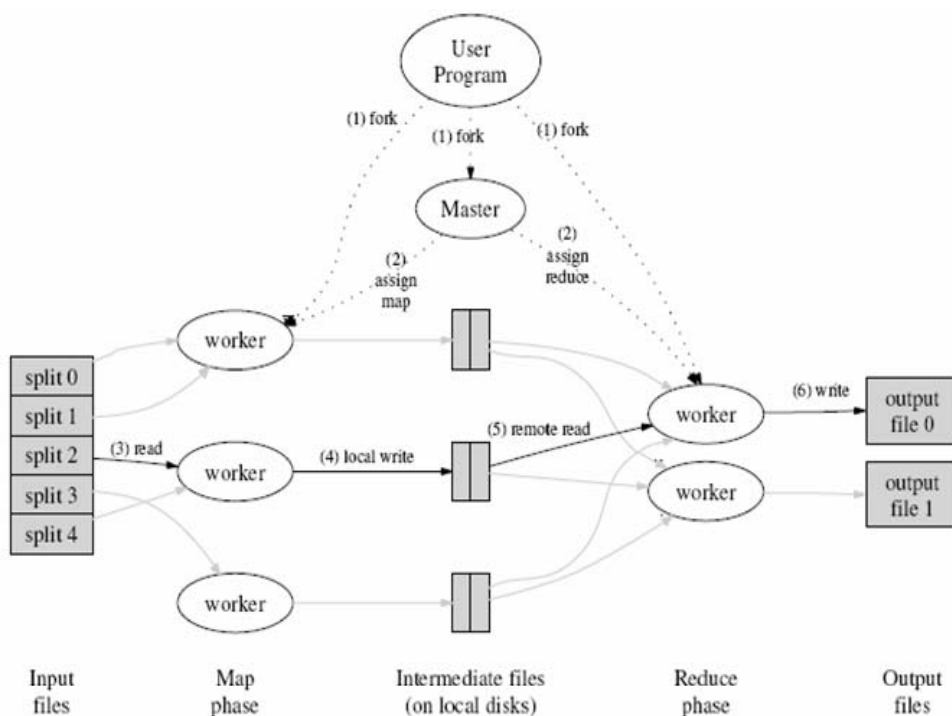


图 3-15 MapReduce 拆分

网络在分布式平台中发挥着通信的作用，计算单元之间的消息交互要通过网络来实现，本章让我们抛开具体的网络硬件设备及华丽的网络术语，从本质上对其进行描述，再来查看在分布式下的变体。

### 4.1 协议分层

计算机真正发挥其普遍性价值效应是在网络出现后，网络在信息技术中扮演的角色是如何将数据从一个世界的一端传到另一端。如前所述，信息世界中的所有内容最终以二进制格式表示。在网络中传递的是二进制数据流，这些看不见的信息经历了不同的线路、网络设备从而到达最后的用户终端。在这些二进制数据中，一部分是企业、用户所关注的业务功能信息，另一部分则是用来进行数据传输管理与控制、保证数据可达的网络功能信息。这一大“块”数据可以分割成不同的“层”，每一层关注特定的功能，交换功能获取其中一层，路由功能获取一层，除此之外为了保证可靠性、进程间的通信，还有传输功能的一层，直到剩下最后应用关注的数据。从上面可以看到，为了实现网络功能（交换、路由、传输等），我们在网络上做两件事：

- 在一个体系模型中分层，定义每一层的功能；
- 对于每一层的功能定义具体协议。

在这一模型中，每个分层都接收由它下一层所提供的特定服务，并且负责为自己的上一层提供特定服务。上下层之间进行的交互常常被称为接口，同一层之间的交互被称为协议。

定义好协议后，可规定好当前层的二进制数据中每一位表示什么。网络数据的传递的过程按照次序在相关节点上实现不同的功能。将一段二进制数据分为不同层级，实现不同功能的形式，也就是常说的协议分层，如图 4-1 所示。

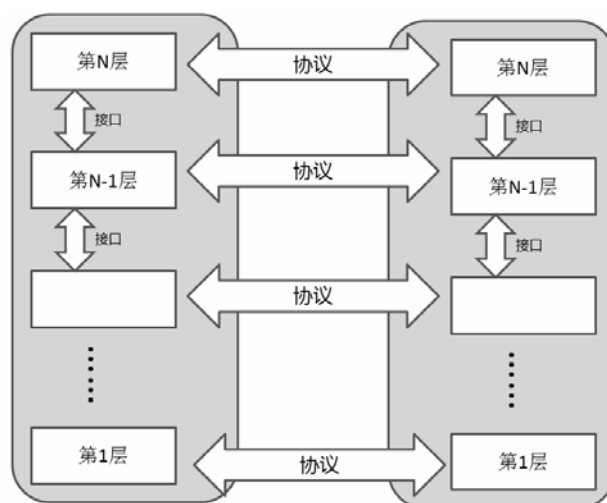


图 4-1 协议分层

### 4.1.1 OSI网络体系模型

OSI（Open System Interconnect，开放式系统互联），一般叫作 OSI 参考模型，它是 ISO（国际标准化组织）组织在 1985 年研究的网络互联模型。该体系结构标准定义了网络互连的七层框架，即 ISO 开放系统互连参考模型。在这一模型下进一步详细规定了每一层的功能。

#### 1. 物理层

提供建立、维护和拆除物理链路所需的机械、电气、功能和规程的特性；提供在传输介质上传输非结构的位流及物理链路故障检测的有关指示。在这一层，数据还没有被组织，仅作为原始的位流或电气电压处理，单位是比特。

#### 2. 数据链路层

负责在两个相邻结点之间的线路上无差错地传送以帧为单位的数据，并进行流量控制。每一帧包括一定数量的数据和一些必要的控制信息。与物理层相似，数据链路层要负责建立、维持和释放数据链路的连接。在传送数据时，如果接收点检测到所传的数据中有差错，那么要通知发送方重发。

#### 3. 网络层

为传输层实体提供端到端的交换网络数据传送功能，使得传输层摆脱路由选择、交换方式、拥挤控制等网络传输细节；可以为传输层实体建立、维持和拆除一条或多条通信路径；对网络传输过程中发生的不可恢复的差错予以报告。

网络层将数据链路层提供的帧组成数据包，包中封装网络层包头，其中含有逻辑地址信息——源站点和目的站点地址的网络地址。

#### 4. 传输层

为会话层实体提供透明、可靠的数据传输服务，保证端到端的数据完整性；选择网络层的最适宜的服务；提供建立、维护和拆除传输连接的功能。传输层根据通信子网的特性，最佳地利用了网络资源，为两个端系统的会话层之间提供建立、维护和取消传输连接的功能，并以可靠和经济的方式传输数据。在这一层，信息的传送单位是报文。

#### 5. 会话层

为彼此合作的表示层实体提供建立、维护和结束会话连接的功能；完成通信进程的逻辑名字与物理名字之间的对应；提供会话管理服务。

#### 6. 表示层

为应用层进程提供能解释所交换信息含义的一组服务，即将要交换的数据从适合某一用户的抽象语法，转换为适合 OSI 系统内部使用的传送语法，提供格式化的表示和转换数据服务。数据的压缩、解压缩、加密和解密等工作都由表示层负责。

#### 7. 应用层

提供 OSI 用户服务，即确定进程之间通信的性质，以满足用户需要，并提供网络与用户应用软件之间的接口服务。

### 4.1.2 OSI与TCP/IP协议簇

人类世界中的理想模型与现实产品是并行交织发展的，在实践中获取宝贵经验，基于经验而完善原有模型，之后再实现新的产品，如此迭代进行，不断发展。我们可以说 OSI 是一套理想的网络模型，它是建立在大量已存在的现实经验上的理想模型。OSI 模型很好地定义了网络中每一层的功能，让其结构清晰明了。值得注意的是，对于 OSI 的每一层功能，OSI 也制订了相应的协议规范，但是在现实世界中没有得到大范围应用。这个应用是指最后各个厂商的硬件、软件遵循这套协议来进行交互。为什么得不到大范围应用呢？最大的原因是 OSI 模型出现的时间太晚，当时 TCP/IP 协议簇已经得到了大量厂商的支持，在无法兼容的情况下短时间内彻底更新为新的协议规范几乎是不可能的；另外 OSI 模型的定义在理想中趋于完美，但在应用、表示、会话三个功能层上太过复杂，或者说它做了太多应该开放给厂商或其他公共机构做的事情，以致硬件设备、软件厂商很难完全遵循它进行实现。

TCP/IP 协议簇的发展是从下至上的，它并没有从一开始就定义好网络的所有功能层级，而是围绕 TCP/UDP、IP 两个主要协议发展出了一套协议栈。它抓住了协议分层的核心思想，IP 协议的下一层协议可以依据不同的软硬件而改变，而在 TCP/UDP 之上，对于不同的应用让其实现自己的协议，不做硬性要求，随之发展了一组得到大量应用的应用协议，例如 HTTP、FTP、SMTP 等。

表 4-1 显示了 TCP/IP 与 OSI 的对应关系。

表 4-1 TCP/IP 与 OSI 的对应关系

TCP/IP 结构对应 OSI			
OSI	TCP/IP	功 能	TCP/IP 协议簇
应用层	应用层	文件传输、电子邮件、文件服务、虚拟终端	TETP、HTTP、SNMP、FTP、SMTP、DNS、Telnet 等
表示层		翻译、加密、压缩	没有协议
会话层		对话控制、建立同步点（续传）	没有协议
传输层	传输层	端口寻址、分段重组、流量、差错控制	TCP、UDP
网络层	网络层	逻辑寻址、路由选择	IP、ICMP、OSPF、ELGRP、IGMP、RIP、ARP、RARP
数据链路层	链路层	成帧、物理寻址、流量、差错、接入控制	SLIP、CSLIP、PPP、MTU
物理层		设置网络拓扑结构、比特传输、位同步	ISO2110、IEEE802、IEEE802.2

4.1.3 交换、选路与传输

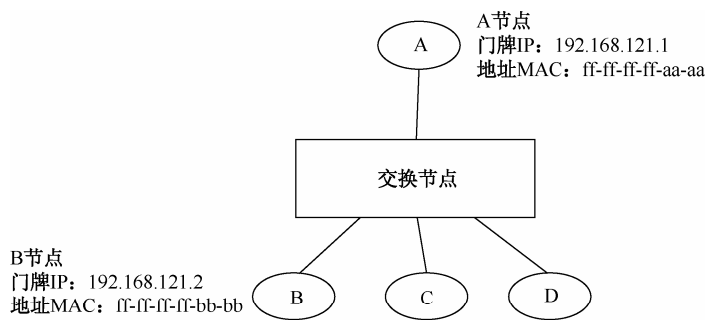
以 TCP/IP 协议栈为例，我们在这一节描述两个节点间进行通信的网络处理逻辑。我们暂时忽略互联互通网络世界里的各种物理设备，仅仅将其抽象为三种：计算节点、交换节点、路由节点，它们要实现世界上任意两个位置的计算节点进程之间的通信。

1. 交换

计算节点是实际的通信对象，最简单的通信方式是直接连接，在两点间直接连接一条网线。但这个世界需要通信的节点太多，假设有 10 个节点相互间需要通信，如果每个节点都需要直接连接，那么需要有 45 (10×9/2) 条线，这显然很浪费资源；如果中间有一个节点承担起数据转发的任务，那么网络连接的拓扑要简单很多，完成这个转发功能的节点就是交换节点。

交换节点负责转发连到其上的计算节点数据，在交换层面上（同一子网内），计算节点的寻址方式是广播。在交换层面上是实现两个相邻计算节点之间的通信，所有以星型结构连接在同一套二层交换设备上的节点可视为相邻，负责互连的二层交换设备可以视为透明。计算节点间点对点的通信是基于物理地址的（即通常所说的网卡 MAC），但实际上用户应用并不在程序中向固定的物理地址发送数据包，而是以一个虚拟的可变的“门牌”地址发起访问，这个地址亦即 IP。计算节点要解决的第一个问题是将 IP 门牌地址转变成固定的可直接通信的物理地址，它会向所有计算节点发出广播，询问“谁有这个门牌，请你应答我”。发现自己的门牌与询问内容相吻合的计算节点会发出响应：“我在这里”，并附上它的物理地址。

这时请求端计算节点获取到可用于直接通信的物理地址，而交换节点记录下了物理地址来自于交换节点中的哪个端口，下一次请求端、接收端的计算节点可以直接通信。如图 4-2 所示。



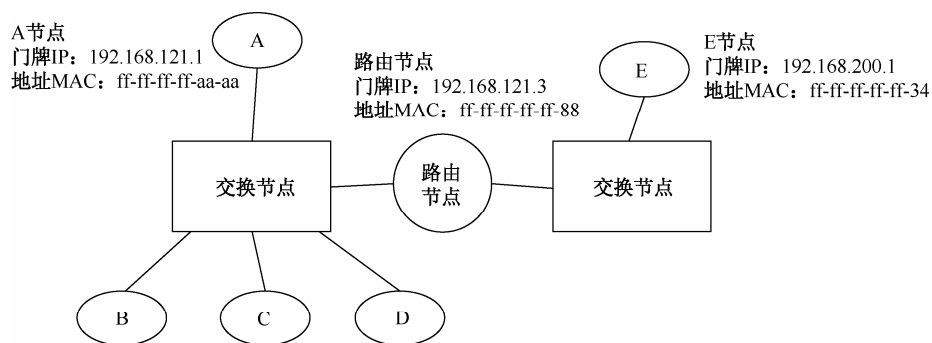
A: “谁有门牌号192.168.121.2?”  
交换节点广播到所有二层计算节点上 (B、C、D)  
B:发现与自己门牌一致, 响应A节点, 发送MAC地址

图 4-2 二层交换节点通信

## 2. 选路

交换节点通过广播完成了相邻节点间的通信。世界上要通信的计算节点的数据量是巨大的，我们可以认为其是无限的，如果每次寻址都要广播通知到所有的节点，那么将造成大量的垃圾通信数据，这显然也是不合理的。我们将这个无限的数据通信节点划分成不同的交换域或者说子网，在内部子网之间利用广播寻址，而在外部的交换域之间使用路由节点进行互连，通过选路完成不同交换域中的计算节点通信。

路由节点连接着两个交换域，计算节点通过目的门牌（IP 地址与掩码）来判断与其是否在一个子网内。当发现不在一个子网时，它会将数据包发送给子网内的路由节点，可称之为网关。网关在收到数据包后，会依据目的门牌进行选路，最终将数据包送达另一个区域的计算节点。如图 4-3 所示。



A节点: 访问E节点192.168.200.1, 若不在一个子网, 则发送给路由网  
A节点: “谁有网关路由门牌号192.168.121.3?”  
交换节点: 广播到所有二层计算节点上 (B、C、D)  
路由节点: 发现与自己门牌一致, 响应A节点, 发送MAC地址  
A节点: 将发送给E的数据包发送到路由节点  
路由节点: 转发数据包

图 4-3 三层路由节点通信

### 3. 传输

当两个计算节点通过交换与选路完成数据通信后，在计算节点上真正使用数据的是操作系统中的进程或者说用户应用。在操作系统上有多个用户应用进程并存，通过传输层，在操作系统中虚拟出唯一的端口号（Port），将数据分发到不同的进程中。

在传输层上，除了通过端口号区分不同的应用程序数据，在端与端间可采用基于面向连接的服务（TCP 协议）。面向连接的服务在每次通信前都将打开、关闭连接，在数据交互过程中，相互之间需要获取一个“确认”信息来保证数据已到达对端，在这个面向连接、相互确认的基础上建立起了一套用于可靠性保证的重传机制、防止拥塞的流量控制机制等。

### 4. 面向连接与无连接服务

不论是在 TCP/IP 还是在 OSI 参考模型中，任意相邻两层的下层为服务提供者，上层为服务调用者。下层为上层提供的服务可分为两类：面向连接的服务和无连接的服务。

#### 1) 面向连接的网络服务

面向连接的网络服务又称为虚电路（Virtual Circuit）服务，它具有网络连接建立、数据传输和网络连接释放三个阶段，是按顺序传输可靠的报文分组方式，适用于指定对象、长报文、会话型传输要求。

面向连接的服务以电话系统为模式。要和某个人通话，则首先要拿起电话，拨号码，通话，然后挂断。同样在使用面向连接的服务时，用户首先要建立连接，使用连接，然后释放连接。连接在本质上像个管道：发送者在管道的一端放入物体，接收者在另一端按同样的次序取出物体；其特点是收发的数据不仅顺序一致，而且内容相同。

#### 2) 无连接的网络服务

无连接的网络服务的两个实体之间的通信不需要事先建立好一个连接。无连接的网络服务有 3 种类型：数据报（Datagram）、确认交付（Confirmed Delivery）与请求回答（Request Reply）。

无连接的服务以邮政系统为模式。每个报文（信件）带有完整的目的地址，并且每个报文都独立于其他报文，由系统选定的路线传递。在正常情况下，当两个报文发往同一目的地时，先发送的先到达。但是，也有可能先发送的报文在途中延误了，后发送的报文反而先到达，而这种情况在面向连接的服务中是绝对不可能发生的。

## 4.2 网络物理设备

网络体系模型中的各种协议是由具体的网络物理要素所实现的，其中包括各种各样的



连接电缆和网络设备。如图 4-4 所示。

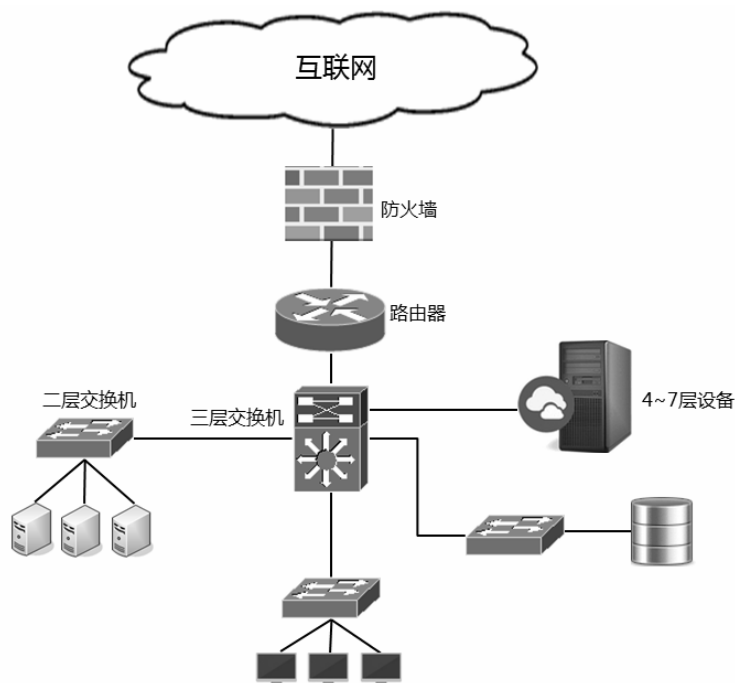


图 4-4 网络构成要素

网络设备遵循协议规范完成相应的网络功能，其包括二层交换机、三层路由器、4~7 层网络设备（防火墙）。

负责各种物理设备连接的是设备上的接口与连线。

### 4.2.1 连线与接口

协议的载体是二进制数，数的物理表现方式是光、电、磁，而光、电、磁的载体则是我们看得见、摸得着的物理设备了。我们将数据中心的网络物理设备抽象成线、接口与设备。

同轴电缆（Coaxial Cable）由 4 层物料构成：最里边是一条导电铜线，线的外面有一层塑胶（用作绝缘体、电介质）围拢，绝缘体外面又有一层薄的网状导电体（一般为铜或合金），最外层的绝缘物料作为外皮。如图 4-5 所示。

双绞线（Twisted Pair）是由两条相互绝缘的导线按照一定的规格互相缠绕（一般以顺时针方向缠绕）在一起而制成的一种通用配线，把两根绝缘的铜导线按一定规格互相绞在一起，可降低信号干扰的程度，每一根导线在传输中辐射的电波会被另一根线上发出的电波抵消。其中外皮所包的导线两两相绞，形成双绞线对，因而得名双绞线。如图 4-6 所示。

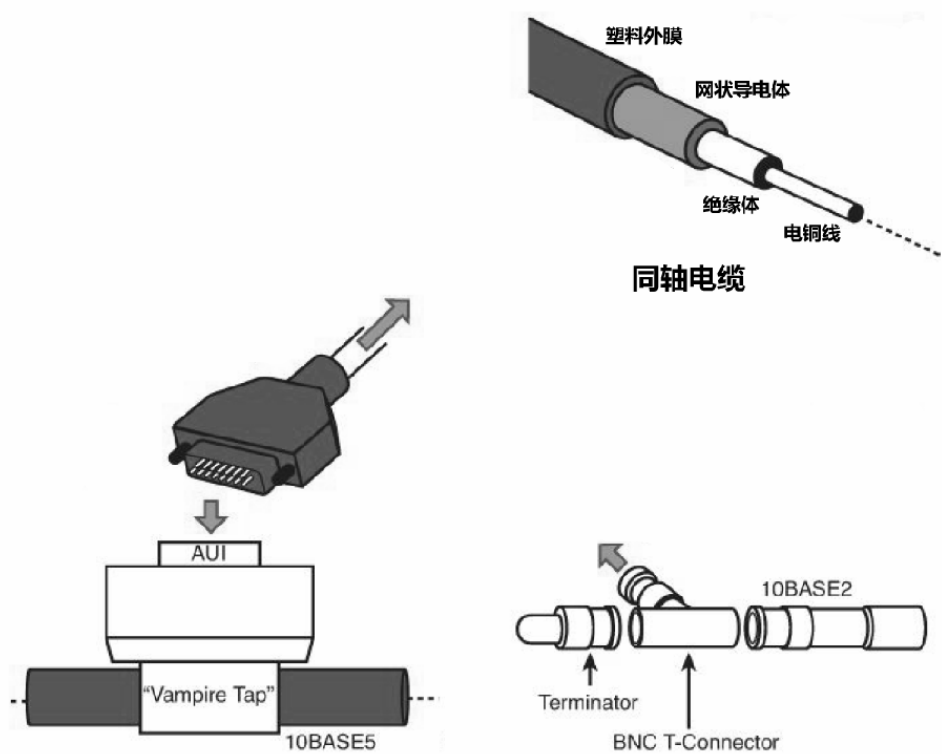


图 4-5 同轴电缆

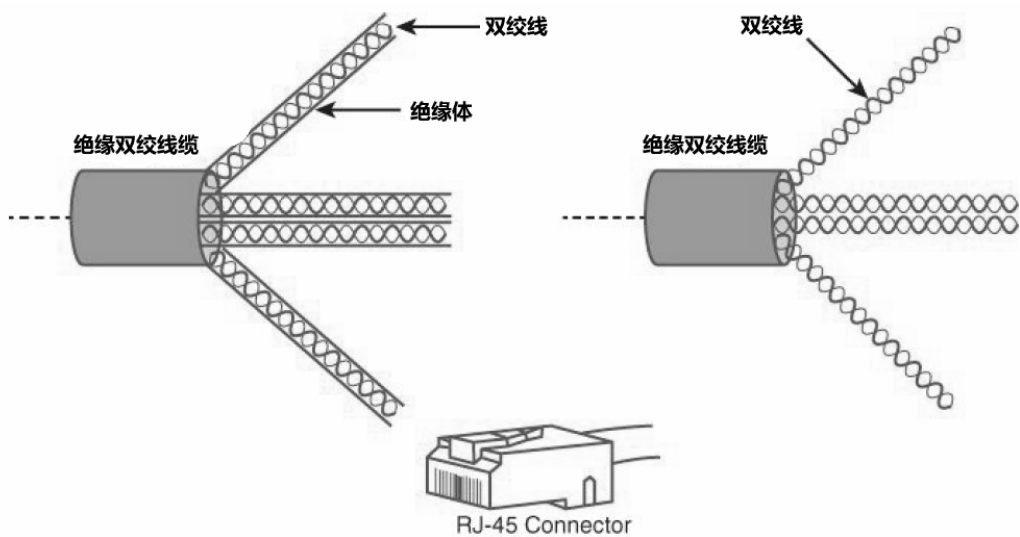


图 4-6 双绞线

光纤（Optical Fiber）是一种导致光在玻璃或塑料制成的纤维中，以全反射原理传输的光传导工具。微细的光纤封装在塑料护套中，使得它能够弯曲而不至于断裂。通常光纤一端的发射设备使用发光二极管或一束激光将光脉冲发送至光纤，光纤另一端的接收设备使

用光敏组件检测脉冲。包含光纤的线缆被称为光缆。由于光在光导纤维中的传输损失比电在电线中的传输损失低得多，更因为主要生产原料是硅，硅的蕴藏量极大，较易开采，价格便宜，所以光纤被用作长距离的信息传递工具。如图 4-7 所示。

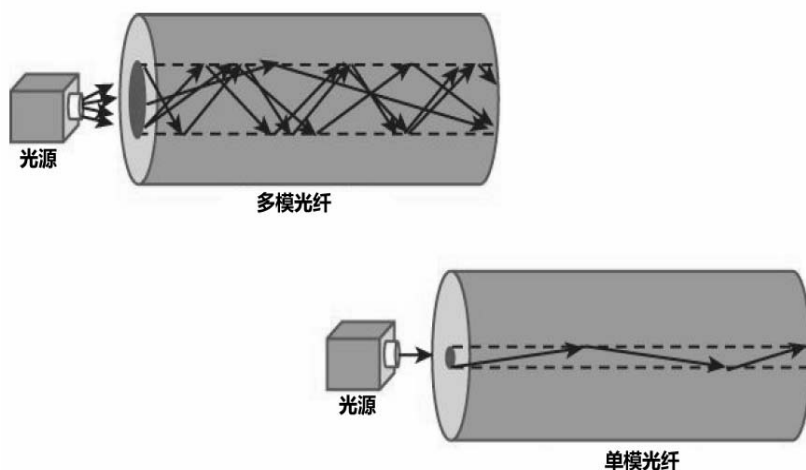


图 4-7 光纤

接口有如下几种。

#### 1) AUI 接口

AUI 接口专门用于连接粗的同轴电缆，在早期的网卡上这样的接口与集线器、交换机相连组成网络，一般用不到。AUI 接口是一种“D”型 15 针接口，之前在令牌环网或总线型网络中使用到，可以借助外接的收发转发器（AUI-to-RJ-45），实现与 10Base-T 以太网络的连接。

#### 2) RJ-45 接口

RJ-45 接口是现在很常见的网络设备接口，俗称“水晶头”，专业术语为 RJ-45 连接器，属于双绞线以太网接口类型。RJ-45 插头只能沿固定方向插入，设置了一个塑料弹片与 RJ-45 插槽卡住以防止脱落。这种接口在 10Base-T 以太网、100Base-TX 以太网、1000Base-TX 以太网中都可以使用，传输介质都是双绞线，不过根据带宽的不同对介质也有不同的要求，特别是与 1000Base-TX 千兆以太网连接时，至少要使用超五类线，要保证稳定、高速，还要使用六类线。

#### 3) SC 光纤接口

SC 光纤接口在 100Base-TX 以太网时代就已经得到了应用，当时被称为 100Base-FX（F 是 Fiber 的缩写），不过当时由于性能并不比双绞线突出成本却较高，因此没有得到普及。业界大力推广千兆网络，SC 光纤接口重新受到重视。

光纤接口的类型很多，SC 光纤接口主要用于局域网交换环境，在一些高性能以太网交换机和路由器上提供了这种接口，它与 RJ-45 接口看上去很相似，不过 SC 接口显得更扁些，其明显区别还是里面的触片，如果是 8 条细的铜触片，则是 RJ-45 接口；如果是一根铜柱，则是 SC 光纤接口。

### 4) Console 接口

在可进行网络管理的以太网交换机上一般都有一个“Console”接口，它是专门用于对交换机进行配置和管理的。通过 Console 接口连接并配置交换机，是配置和管理交换机必要步骤。因为其他方式的配置往往需要借助于 IP 地址、域名或设备名称才可以实现，而新购买的交换机显然不可能内置这些参数，所以 Console 接口是最常用、最基本的交换机管理和配置接口。

## 4.2.2 二层交换机

交换机工作于 OSI 参考模型的第二层，即数据链路层。这种只具备“交换”功能的交换机通常被称为接入交换机。交换机内部的 CPU 会在每个接口成功连接时，通过将 MAC 地址和接口对应，形成一张 MAC 表。在今后的通信中，发往该 MAC 地址的数据包将仅送往其对应的接口，而不是所有的接口。因此，交换机可用于划分数据链路层广播，即冲突域，而不能划分网络层广播，即广播域。

交换机与集线器的不同之处是，集线器会将网络内某一用户发送的数据包传至所有已连接到集线器的电脑。而交换机只会将数据包发送到指定目的地的电脑（通过 MAC 表），相对能减少数据碰撞及数据被窃听的可能性。交换机更能将同时传到的数据包分别处理，而集线器则不能。如图 4-8 所示。



图 4-8 思科 Catalyst 3850 系列堆叠式接入交换机

## 1. VLAN

当交换机处理二层广播包时，它会将数据包转发到所有的接口上，如果有一种方式来建立一个范围较小的广播组，各组成员在组内进行广播通信，那么这将过滤掉大量无关的数据包，降低设备的总负载。虚拟 LAN 或者 VLAN 指的是在一台交换机内提供的不同逻辑的 LAN 的虚拟隔离，对于来源于一个 VLAN 的数据包来说，交换机是无法将其传输到另一个 VLAN 的。

## 2. Trunk

Trunk 是能够同时为多个 VLAN 传递数据包的接口，Trunk 可以用来连接两个交换机设备。我们可以认为 Vlan 会在数据包上打上一个 tag，用来辨识它可以发往那些有同样 Vlan 的接口。而 Trunk 接口在交换机内部可以接收来自多个 Vlan tag 的数据包。而在交换机外部，这个 Trunk 接口通常连接的是另外的设备，可能是交换机，也可能是计算节点，这时对端设备如果想通过该 Trunk 接口传输数据，那么它必须提前打上对应的 Vlan tag。

## 3. 生成树

交换机只要收到广播包，就会快速地在每个接口上复制（与源广播包相同的 Vlan），在一个环路的二层网络环境中，广播会被无止境地复制，其结果被称为广播风暴，整个网络会因此而中断、不可用。在 LAN 中，生成树用于确保二层无环路产生。

生成树的一般算法如下：首先进行根网桥的选举，其依据是网桥优先级（Bridge Priority）和 MAC 地址组合而成的桥 ID，桥 ID 最小的网桥将成为网络中的根桥（Bridge Root）。在此基础上，计算每个节点到根桥的距离，并由这些路径得到各冗余链路的代价，选择最小的路径作为通信路径（相应的端口状态变为 forwarding），其他的路径就成为备份路径（相应的端口状态变为 blocking）。在 STP 生成过程中的通信任务由 BPDU 完成，这种数据包又分为包含配置信息的配置 BPDU（其大小不超过 35B）和包含拓扑变化信息的通知 BPDU（其长度不超过 4B）。

## 4.2.3 路由及三层交换

路由器属于 OSI 的第三层，专门用于连接两个以上的二层网络区域。通常具备路由功能的网络设备同时拥有交换能力，许多路由器加入到局域网内执行与交换机相同的功能，我们统称这种网络设备为三层交换机。

网络 OSI 三层的数据通信，节点之间通过路由表中的策略来接收与发送数据包。在服务器上，我们通常人工设置这些路由策略（可称为静态路由）。在局域网等规模较小的网络中，人工设定路由策略没有任何问题，但在一个具有较大规模的网络中，人工指定路由策略，将会带来巨大的工作量，并且管理、维护路由表变得十分困难。为了解决这个问题，动态路由协议产生了，动态路由协议可让路由器自动学习到其他路由器的路由信息，并且

在网络拓扑发生变化时自动更新路由策略。

常见的动态路由协议有 RIP、IGRP（Cisco 私有协议）、EIGRP（Cisco 私有协议）、OSPF、IS-IS、BGP 等。RIP、IGRP、EIGRP、OSPF、IS-IS 是内部网关协议（IGP），适用于单个 ISP 的统一路由协议的运行。BGP 是自治系统间的路由协议，是一种外部网关协议，多用于在不同的 ISP 之间交换路由信息。如图 4-9 所示。



图 4-9 思科 Nexus 7710 三层交换机

#### 4.2.4 四～七层网络设备

四～七层网络设备负责处理 OSI 模型中从传输层到应用层的数据，以 TCP 及其以上的应用协议数据为基础对其进行解析与处理。在这个层面上的设备分支类型非常丰富，涉及安全管控的防火墙、性能扩展的负载均衡，以及其他各种应用层功能支持（页面缓存、证书加密、网络加速等）。如图 4-10 所示。



图 4-10 F5 负载均衡 big ltm 6900

### 4.2.5 现实网络构成

我们以交通道路为例说明现实网络的构成。在大型城市的道路交通网络中使用高速公路来帮助人们在不同城市、省份之间来往。在计算机网络中类似于高速功能的部分叫作“骨干”，它们是计算机的中心，通常使用高性能路由器相互连接使之快速传递大量数据。网络中相应于高速公路出入口的部分叫作“边缘网络”，常用设备为多功能路由器和三层交换机。高速公路的出入口通常连接的是国道、省道，从而可以通往市区的各个位置。计算机网中连接“边缘网络”的部分叫作接入层、汇聚层。

中国的主要网络运营商有中国电信、中国联通，以及其他二级运营商（中国移动、中国铁通、教育网），在骨干网上占主要份额的是中国联通、中国电信，其他二级运营商拥有10GB级别的若干骨干网络。

## 4.3 网络逻辑拓扑

最通用的数据中心拓扑是三层结构，叫作核心-汇聚-接入（Core-Aggregation-Access）。这种分层拓扑结构增强了网络的模块化、灵活性与弹性，被很多数据中心采用。三层结构如图4-11所示。

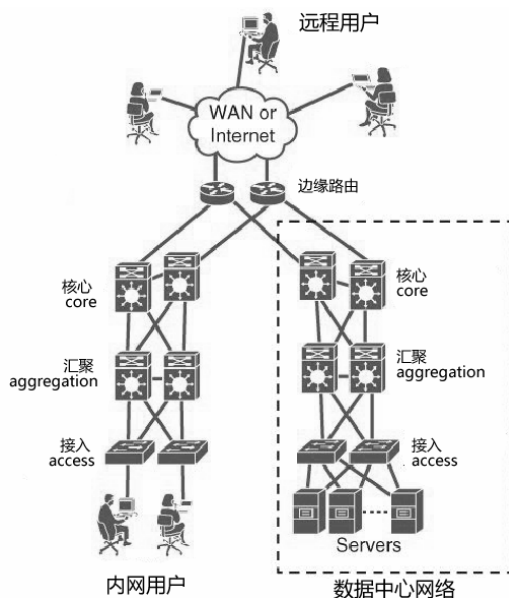


图 4-11 网络三层结构图

企业连入的计算节点可以分为内网办公区域与数据中心区域，办公区域接入的计算节点供内部办公使用，而在数据中心放置的是面向用户的服务器设备，其服务可靠性要求与

办公区不在一个级别上。办公区域的网络设备故障往往只会影响部分用户，而一旦数据中心发生故障，则意味着所有的用户无法访问，企业的全部业务面临中断。计算节点的接入商，其办公桌面仅仅连入一条线即可使用，而服务器会接入两条进行冗余。在数据流向上二者也是截然相反的。

数据中心的三层拓扑得到了大量实际环境的验证。核心层提供数据中心进出流量的交换能力。特别的路由设置功能，可保证多个汇聚交换机之间的连通性。

在汇聚层管辖着自己名下的所有计算节点，它们属于同一个子网，或者说这些子网的所有节点具有同一功能。汇聚层所管辖的区域为满足特定功能要求的安全区域。这些安全区域可以按照大业务模块划分，也可以按照网络功能划分。非军事管理区（DMZ）、合作伙伴区（Partner）、服务区（Server Farm）是常用的划分方式。每一个区域通过一个汇聚交换机上联到核心。在 SF 中放置需要严格的安全控制的核心服务，为了保护 SF 内核心服务的安全性，针对外网用户、合作伙伴用户另外开辟了 DMZ、合作伙伴区域。在汇聚层还会放置防火墙、负载均衡等全局性的网络设备。

接入层直接与服务器相连，其由二层交换机组成，负责子网内的服务器的连通性。

固定的三层架构模型并不适用于所有数据中心，它为具体的环节提供了一般的指导性。如果数据中心的计算节点不多，那么可以将核心与汇聚层合并到一起。相应地，如果数据中心的计算节点量非常大，那么我们可以放置多个核心层。

## 4.4 对网络拓扑的考虑

在设计一个数据中心网络时，对一些非专业技术方面的因素必须提前考虑到，例如数据中心计算节点的增长，未来有多少服务器，需要多少接口、用户量多少等，从而避免 IDC 成为应用瓶颈。

应用带宽在网络设计中也需要重点考虑到，我们用收敛比的概念来说明，收敛比即分配给所有用户的资源与实际可用最大资源的比例。举个例子，超市有 100 个用户同时要求买单，在超市的出口设置了 10 个买单的柜台，是 10:1 的收敛比例。收敛带来的好处是能节省共用设备。在数据中心网络设计中，收敛比体现在如何设置各层交换设备之间的带宽来保证数据流量的通畅。例如，接入层交换机有 32 个 10GB 的以太网服务器接口接入，8 个 10GB 的上联以太网接口，它的收敛比是 4:1。经过测试、调优，可依据应用类型设置合适的收敛比来支持应用流量的需求。

业务系统的相关性也会直接影响到网络拓扑的设计，这主要体现在非技术方面，金融类的应用经常要满足监管机构的要求，与不相干的业务进行隔离或者要求故障域足够小，这样，单独的一套三层物理结构可能就不再满足要求了。



为了保证应用的可靠性、可用性，可对服务器加入冗余的网卡并接入不同的接入层交换机，以避免单点故障。网络在出现连接异常时必须能够快速反应，进行主备切换、恢复异常。

## 4.5 对物理布线的考虑

在数据中心的网络设计中不仅包括正确的逻辑拓扑，网络设备和服务器的放置方式对一个数据中心的生产效率也有很大的影响。美国国家标准学会（ANSI）2005 年批准颁布的《数据中心电信基础设施标准》由美国电信产业协会和 TIA 技术工程委员会编写，其缩写为 TIA-942，包含了机房基础设施组件的规划与设计。

我们重点关注 TIA-942 在基础设施空间布局上的最佳实践，它定义了下面主要的空间与综合布线元素。

- 服务器机房（Computer Room）：容纳所有服务器设备的空间。
- 入口房间（Entrance Room）：在数据中心综合布线系统和建筑外部布线（服务提供商、合作伙伴）的连接处。它应被放置在计算机机房的外部，从而避免物理上的安全缺口。
- 主要分布区（Main Distribution Area, MDA）：在计算机机房中，它是综合布线的中心节点，由于其能够灵活地连通数据中心的其他节点，所以网络核心设备常放置于此。
- 水平分布区（Horizontal Distribution Area, HDA）：也被放置在计算机机房内，在这个区域内集中放置汇聚层网络设备，负责连接接入层的计算机节点，以及网络核心。
- 设备分布区（Equipment Distribution Area, EDA）：在这个区域内放置计算设备及接入网络设备，外部线路与汇聚交换机相连。为了更好地散热，机架接入了冷、热通道。
- ZONE 分布区（Zone Distribution Area, ZDA）：为在 HDA 和 EDA 区之间可选的内部连接点，目的是增强机房空间布局的可重构性、灵活性。

机房布局如图 4-12 所示。

数据中心的布线架构同样会影响到 IDC 的整体管理效率。在还没有形成规模且服务器上架零散时，对于布线架构不会有太多规划要求。在特定区域放置服务器、网络设备并将节点进行互联，在物理位置受限时，可通过跳线连通解决问题。这种直连方式简单明了，交换机接口的使用率也非常高，但其缺点非常明显：当服务器上架达到一定规模时，这种布线方式会严重影响上架效率，上一台服务器都要依据现场情况调整线路，在大规模布线

需求下甚至会出现机柜、地板下空气流动受限的情况。服务器上架之后的运维管理也会因为区域的零散而增加运维管理的复杂度。

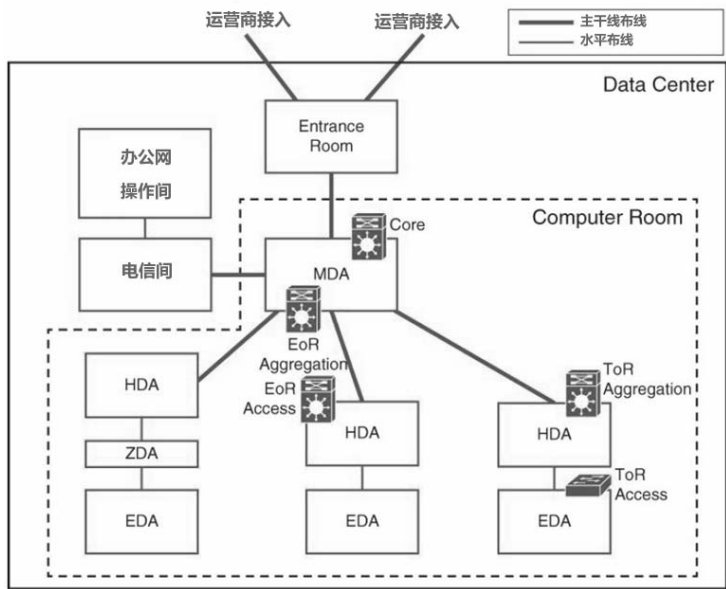


图 4-12 机房布局

为了提升大规模机器设备的上架效率，在云计算的发展趋势下，互联网公司都在采用机架式整机柜上架的方式，在这里引入两种流行的布线方式：ToR（Top-of-Rack）和 EoR（End-of-Row）。

数据中心机房平面布局通常采用矩形结构，为了保证制冷效果，通常将 10~20 个机柜背靠背并排放置成一行，形成一对机柜组（又称为一个 POD），POD 中的机柜都采用前后通风模式，冷空气从机柜前面的板吸入并从后部排出，由此在机柜背靠背摆放的 POD 中间形成“热通道”，相邻的两个 POD 之间形成“冷通道”。热通道正对 CRAC（机房空调），热空气沿热通道流回 CRAC，开始新的一次循环。如图 4-13 所示。

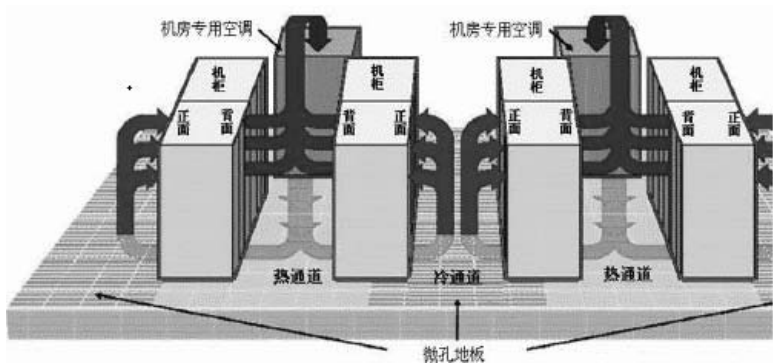


图 4-13 机房制冷散热布局

### 1. 交换机 EOR (End of Row) 布线方式

采用交换机 EOR 布线方式时，每个 POD 中的两排机柜的最外端摆放了两个网络机柜，POD 中所有的服务器机柜都安装了配线架，配线架上的铜缆延伸到了 POD 外端的网络机柜，在网络机柜中接入交换机。机架式服务器安装在服务器机柜中，服务器网卡通过跳线（铜缆）连接机柜中的配线架。

交换机 EOR 布线的方式最为常见。通常在服务器和接入交换机安装之前，已经完成了从服务器机柜到网络机柜的布线施工，设备（服务器/交换机）安装和跳线工作都在服务器机柜内和网络机柜内进行。

如果每台机架式服务器的功率为 500W，且每个服务器机柜的电源输出功率按 4kW 或 6kW 计算，则一个 42U 高度的服务器机柜能安装 8~12 台机架式服务器。

EOR 布线方式的缺点：从服务器机柜到网络机柜的铜缆较多（20~40 根），且距网络机柜越远的服务器机柜的铜缆，在机房中的布线距离越长，由此导致线缆管理维护工作量大、灵活性差。

交换机 MOR (Middle of Row) 布线是对 EOR 布线方式的改进。MOR 方式的网络机柜部署在 POD 两排机柜的中部，由此可以减少从服务器机柜到网络机柜的线缆距离，简化线缆的管理、维护工作。如图 4-14 所示。

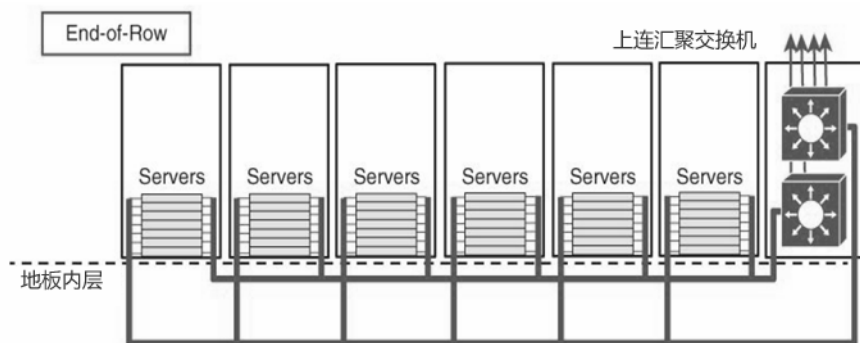


图 4-14 交换机 EOR

### 2. 交换机 TOR (Top of Rack) 布线方式

交换机 TOR 布线方式是对 EOR/MOR 方式的扩展，在采用 TOR 布线时，POD 中每个服务器机柜的上端部署了 1~2 台接入交换机，机架式服务器通过跳线接入机柜内的交换机上，交换机的上行接口通过铜缆或光纤接入 EOR/MOR 的网络机柜中的汇聚交换机上。

TOR 布线方式的特点如下。

- TOR 布线方式简化了服务器机柜与网络机柜之间的布线，从每个服务器机柜到 EOR/MOR 的网络机柜的光纤或铜缆数量较少（4~6 根）。

- 机柜中服务器的密度较高。对于标准的 19 英寸宽、42U 高的机柜，如果采用交换机 TOR 布线方式，则每个机柜可部署 15~30 台 1U 高度的机架式服务器（具体数量需要考虑单台服务器的功耗和机柜的电源输出功率）。

TOR 布线的缺点为：每个服务器机柜受到电源输出功率的限制，可部署的服务器数量有限，导致机柜内交换机的接入接口利用率不足。在几个服务器机柜之间共用 1~2 台接入交换机，可解决交换机接口利用率不足的问题，但这种方式增加了线缆管理的工作量。

从网络设计上考虑，TOR 布线方式的每台接入交换机上的 VLAN 量不会很多，在网络规划时也要尽量避免使一个 VLAN 通过汇聚交换机跨多台接入交换机，因此在采用 TOR 布线方式的网络拓扑中，每个 VLAN 的范围不会太大，包含的接口数量不会太多。但对于 EOR 布线方式来说，接入交换机的接口密度较高，在最初设计网络时，就可能存在包含较多接口数量的 VLAN。

TOR 方式的接入交换机数量较多，EOR 方式的接入交换机数量较少，所以 TOR 方式的网络设备的管理、维护工作量较大。如图 4-15 所示。

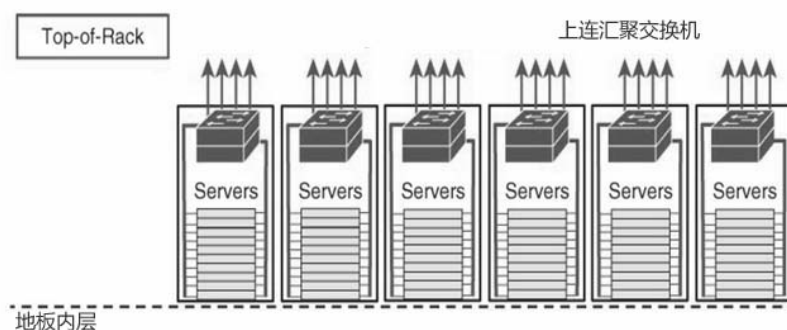


图 4-15 交换机 TOR

## 4.6 网络虚拟化与SDN

网络虚拟化作为一个单独概念并没有存在多长时间，但其背后的技术发展却有了相当长时间的积淀，可以说网络的虚拟化从来没有停止过。人们早就意识到了网络服务与硬件解耦的必要性，也因此诞生了很多技术，我们最熟悉的就有：VLAN（虚拟局域网，工作在二层）、VPN（虚拟专用网络，建立隧道）、VRF（虚拟路由转发）等。

VPN 是指在公共网络上建立起来的“虚拟”专用网络，它的任意两个节点之间并无传统专用网络所需要的端到端的物理链路，而是构建在公用网络供应商所提供的物理网络之上，通过隧道技术实现站点之间的互联，以达到共享物理网络资源的目的，所以说它是一种逻辑网络。VPN 通常用于一些组织或者公司来互连其子部门，也可以用于个人远端接入

公司内部网络。

VRF 允许多个路由表实例在一台物理路由设备上存在，不同物理接口接入的数据流使用不同的路由表，从而将一台物理设备“虚拟”成多台路由器。

我们可以看到 VLAN、VPN 及 VRF 网络虚拟化都是在解决某些具体问题，欠缺完整的技术体系与合理的组织结构，理想模型与技术在一个并行交织的发展过程中实现。现在的网络虚拟化一方面借助于云计算的平台来全面展示其优异的性能；另一方面整合自身的体系结构，以平台化的方式提供更为友好、全面的服务。软件定义网络（SDN）的兴起也为网络虚拟化提供了一个新的思路，如何利用 SDN 去实现网络虚拟化已然成为了一个热门话题。

SDN（Software Define Network）的概念出来很久，被媒体炒得火热。其强调以下两方面的能力。

（1）控制及数据平面分离：传统网络设备紧耦合的网络架构被分拆成控制和数据两个平面。同时，在控制平面增加了集中控制器进行整体调度，将命令和逻辑发送到硬件（路由器或交换机）的数据转发平面。

（2）开放 API 及软件定义：即通过基于 SDN 技术的对外开放的 API 进行软件编程，实现整个网络集中的管理能力，而不需要在每个路由器或交换机上分别以设备为中心进行管理。

网络的使用将像 OS 一样灵活。网络设备像之前的小型机、大型机一样受到 x86 PC 的冲击，封闭可以带来丰厚的利润，但随着竞争的引入又不得不开放。

这里的“封闭”的含义如下。

（1）软件与硬件的强绑定：在网络领域对软硬件的“保护”要比对操作系统的保护做得好很多，网络的操作系统（catOS、IOS）与硬件设备是紧密结合的，并且 Network 还有一个模块的概念。什么是模块？我要加一个新功能，就得买一个模块，插入之前的插槽中，软件功能模块和硬件是联系在一起的，Cisco 的 sales 会说：“我给你一块板子”，你就有了这个功能。

（2）协议、命令行的私有化：一个大型公司的网络系统管理员必须掌握不同网络设备、不同版本的各种命令、配置，以至于为了保存一个配置需尝试 save、write、commit、copy running-config startup-config 等各种命令，更不用说进行各种防火墙配置了。各大厂商为了保持竞争力，对命令行、协议进行保护，越来越专有化，越来越“封闭”。

（3）没有可编程的 API：让 Linux 系统管理员最感到幸福的莫过于在所有的 OS 上有一套统一的 API，不管你是 open、close、socket，虽然有一些兼容性问题，但至少可用、开放。你可以在上面再包裹一层来实现一致性，也可以使用各种配置管理工具（Puppet、salt）来统一标准、集中管理。对于开发人员来说，各大互联网巨头在应用层就已提供了

service 接口，可编程的 API 貌似已经成为这个时代的基本需求。

由于网络设备的封闭性，在互联网浪潮下人们对可编程的需求又是如此迫切，SDN 应运而生。既然谁都可以买一堆计算机，然后加入一群软件工程师中去做出一些了不起的东西，那么网络也理应如此。理想的网络架构应该具备计算机的灵活性，可适用于任何网络硬件，换句话说，应该有一种通用网络操作系统，在此平台下，硬件只负责收发网络数据包，由软件负责思考。如图 4-16 所示。

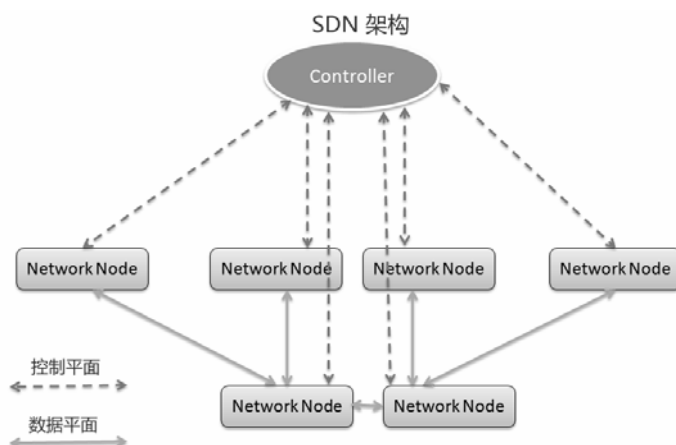


图 4-16 SDN 架构

# 存储资源

存储系统在整个计算机系统中扮演着数据持久化的角色，数据以电、磁、光的物理形式在各个节点中流动，最后在存储系统中“沉淀”。在本章中让我们看看存储资源的组成与运行原理。

## 5.1 俯瞰存储系统

### 5.1.1 数据存储功能分类

在计算机科学的发展历史上，数据存储技术因其不同的功能而被分为广为人知的三类。

#### 1. 主存（Primary Storage）

通常简称为存储器，该层的存储器与 CPU 直接连通，CPU 会不断读取存储在这里的指令集，并在需要时运行这些指令集。在第一级存储器的内部，除了主存（随机存取存储器），还有两个或两个以上的子层。

寄存器被设置在处理器内。每个寄存器都存储数据中的一个字（一个字的大小通常是 32 位或 64 位）。中央处理器内的指令能让算术逻辑单元运行各种计算或者处理数据。寄存器传输数据的速度是所有存储器中最快的。

CPU 缓存传输数据的速度仅次于寄存器，被用来提升电脑的性能。大部分经常被使用的数据（存在主存中）会被复制一份存在高速缓存内，这样可提升速度，否则速度将会大幅降低，且寄存器也容纳不下那么多数据。多层缓存结构也经常被使用，第一层缓存容量最小、速度最快且位于处理器内部；第二层缓存容量较大且较慢。

#### 2. 二级存储（Secondary Storage）

又称外部存储器或辅助存储器。和主存不同的是，二级存储器和 CPU 并没有直接连通，计算机经常使用存储器的 I/O 通道来与之连接，二级存储器使用数据缓冲器来将数据发

送至第一级存储器。在不供应电源的情况下，第二级层储器的数据仍然不会消失，表示它是持久化的。硬盘被广泛地作为第二层存储器，硬盘访问数据的时间大约是几千分之一秒，或者几个毫秒。然而，随机存取存储器访问数据的时间仅为几十亿分之一秒，或者几个纳秒。硬盘的速度只有存储器访问速度的百万分之一。

### 3. 三级存储（Tertiary Storage）

这是指可直接插入计算机或从计算机中拔除的存储设备，里面的数据在被使用前通常都会复制到二级存储器内。该款存储器的访问速度比二级存储器要慢得多（前者为 5~60 秒，后者为 1~10 毫秒）。这款存储器的优势在于其拥有庞大的存储空间，典型的例子为磁带柜和光学记录库。

本章所讲述的存储以二级存储为主。在这个层面上因新技术引入、产品多样化及虚拟化而变得复杂、难以理解。我们试图找到存储的“主干”，还原其本来面貌，清晰地把握住技术之间的关联性，之后在每个节点上进行详细的介绍。

## 5.1.2 文件存储的三个层级

我们在打开一个文件，并对其写入内容时，这是一次存储的写操作。其过程为：应用程序发起文件操作请求，调用操作系统的系统调用（System Call），陷入内核模式，在统一的虚拟文件系统的接口下按特定文件系统的格式准备数据，之后调用设备驱动程序，向物理块设备发起 I/O 请求。整个过程如图 5-1 所示。

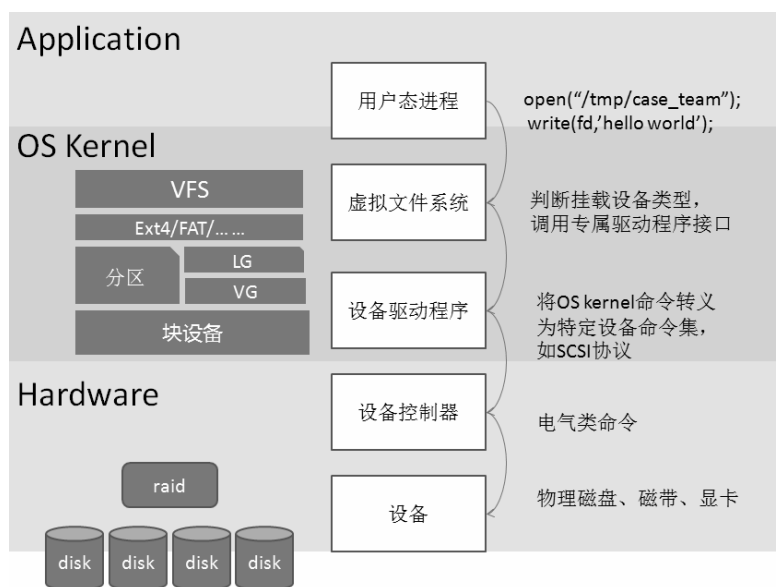


图 5-1 应用程序打开、写入文件的过程



我们将这个过程从下到上分为以下三个层级。

### 1. 第一层：磁盘

在这个层级上我们可以看到最终存储实体。无论其设备类型、技术参数有何差异，我们都会在这个存储实体中读取或者写入二进制数据，我们抽象地将它称为磁盘。计算机在与磁盘通信时，节点之间的交互都有其遵循的协议规范，SCSI、ATA 是当前使用最为普遍的两个 I/O 通信协议。

如果把存储通信看作用户需求，把 I/O 通信的协议栈映射到 OSI 网络参考模型中，那么 SCSI、ATA 协议会在栈的顶层。而在栈的底层可以选择其他不同的协议，这样的存储操作可以在不同的物理设备、网络上运行。例如选择 FC 协议簇则可以让存储运行在光纤通道网中，选择 TCP/IP 协议簇则可以让存储运行在以太网中。

在存储端，我们使用磁盘存储数据，使用 SCSI 或 ATA 协议进行通信。在计算节点端使用主板集成接口或者专门的适配卡与磁盘相连，这些接口包括：

- 遵循 ATA 协议的 IDE 接口、SATA 接口；
- 遵循 SCSI 协议的并行 SCSI 接口、串行 SCSI (SAS) 接口；
- 遵循 SCSI 协议的可以用来承载 FibreChannel 协议的串行 FC 接口。

计算节点有与磁盘进行沟通的硬件接口，而在操作系统软件层面上，它需要专门用于 I/O 控制器、适配于特定硬件接口的驱动程序，驱动程序掩盖了硬件间的差异性，提供一致的接口让上层调用。对于各种底层硬件来说，经过驱动程序的转换，操作系统看到的都是标准的块设备，第一层到此结束。

第一层虽然到此结束，但从操作系统来看，存储实体提供块设备服务。从存储的发展来看，在存储实体内部发生了很大的变化，它可以使用硬件 RAID 卡，将多个磁盘抽象成一个，提高性能与可用性，它也可以运行在各种存储网络架构中，满足不同场景的需求。

### 2. 第二层：卷管理

计算机系统看到的是块设备，在第二层它将做些什么呢？该层的主要任务是为具体的文件系统做格式化准备，“合久必分，分久必合”是这个层级的主体思想，我们也可以将这个层级的任务称作卷管理。在生产服务器环境中，操作系统看到块设备后会有两种做法：将一个块设备切分成多个分区（个人桌面电脑常使用的方法）；或者将多个块设备合并成一个卷（逻辑卷管理）。

#### 1) “合久必分”的磁盘分区

对于桌面电脑而言，个人用户并没有要求一个连续的、大容量的存储卷需求，相反，用户需要根据文件的类型将其保存到不同的卷中，这样便于硬盘的规划、文件的管

理，以及提高系统运行效率。操作系统会对块设备进行分区，包括主分区、扩展分区、逻辑分区。

主分区也叫作引导分区，是独立的，对应磁盘上的第一个分区。

除主分区外，剩余的磁盘空间就是扩展分区了。扩展分区是一个概念，实际上是看不到的。当整个硬盘被分为一个主分区时，就没有了扩展分区。

逻辑分区在扩展分区上面，可以创建多个逻辑分区。逻辑分区相当于一块存储的截止，与操作系统的其他逻辑分区、主分区没有关系，是“独立的”一块卷。

### 2) “分久必合”的逻辑卷管理

对于服务器而言，其上运行着一个或多个应用程序，其数据存储的容量需求与桌面电脑相比有不可预知性，若应用请求量突增，则存储容量也随之而增。另外，当卷满时，桌面电脑用户可以随时进行文件整理，移除或者移动相关文件以释放空间。但对于在服务器中的运行着的生产应用而言，当卷满时，它希望采用的是一种“透明”的方式进行扩容，而不影响应用的运行。

逻辑卷管理是建立在物理存储设备之上的一个抽象层，它将所有物理块设备聚合成一个大的卷组，将逻辑卷分配给应用，随后可以灵活地进行卷容量管理，在不用停应用、下载文件系统的情况下即可完成卷的扩容。

## 3. 第三层：文件系统

在做好第二层准备后，即可对卷进行文件系统格式化，并进行卷挂载。

文件系统是对一个存储设备上的数据和元数据进行组织的机制，它解决如何在存储设备上存储数据，包括存储布局、文件命名、空间管理、安全控制等。主要的文件系统类型有 ext4、ext3、NTFS、FAT 等，我们可以选择其中一种对卷格式化，在 Linux 操作系统上会有一个卷挂载（mount）的动作，它将一个文件系统附着到当前文件系统的层次结构中，随后应用的用户即可使用文件存储。

在将数据存储到不同类型的文件系统时，还有另一个问题需要回答：为什么我们可以使用一致的系统调用（创建、打开、写入、读取、关闭）来操作不同的文件系统？这种差异化问题是由操作系统的虚拟文件系统（VFS）解决的（见图 5-2），VFS 是操作系统内核和具体 I/O 设备之间封装的一层通用访问接口，通过这层接口，内核可以以同一种方式访问各种 I/O 设备。它简化了应用程序的开发，应用通过统一的系统调用访问各种存储介质；也简化了新文件系统加入内核的过程，新文件系统实现了 VFS 的各个接口即可，不需要修改内核部分。

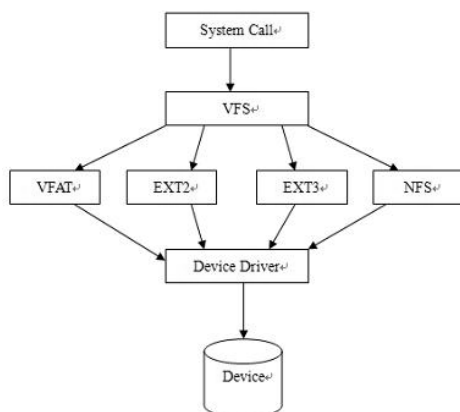


图 5-2 虚拟文件系统

下面让我们以文件存储的三个层级为主轴，对其中的关键技术展开讨论。

## 5.2 磁盘与磁盘阵列

### 5.2.1 硬盘的物理构造

硬盘的物理结构一般由磁头（Head）与碟片、电动机、主控芯片与排线等部件组成，当主电动机带动碟片旋转时，副电动机带动一组磁头到相对应的碟片上并确定读取正面的碟面还是反面的碟面。磁头悬浮在碟面上画出一个与碟片同心的圆形轨道（磁轨或称柱面），这时由磁头的磁感线圈感应碟面上的磁性，并使用硬盘厂商指定的读取时间或数据间隔定位扇区，从而得到该扇区的数据内容。硬盘的物理结构如图 5-3 所示。

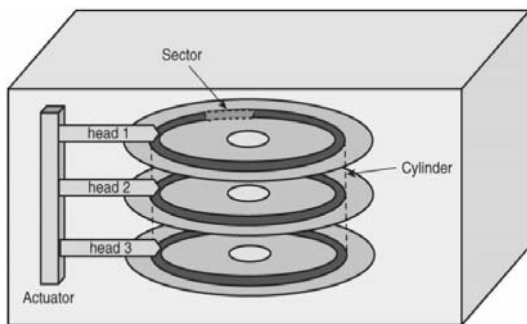


图 5-3 硬盘的物理结构

#### 1) 磁道

磁盘旋转时，若磁头保持在一个位置上，则每个磁头会在磁盘表面划出一个圆形轨迹，这些圆形轨迹就叫作磁道（Track）。

### 2) 柱面

柱面为在由多个盘片构成的盘组中，由不同盘片的面但处于同一半径圆的多个磁道组成的一个圆柱面（Cylinder）。

### 3) 扇区

磁盘上的每个磁道被等分为若干个弧段，这些弧段便是硬盘的扇区（Sector）。硬盘的第一个扇区叫作引导扇区。

硬盘的一次读写动作包括寻道、旋转和数据传输时间。

### 4) 寻道时间（Seek）

是指将读写磁头移动至正确的磁道上所需要的时间。寻道时间越短，I/O 操作越快，目前磁盘的平均寻道时间一般为 3~15ms。

### 5) 旋转延迟（Rotation）

是指盘片在旋转中将请求数据从所在扇区移至读写磁头下方所需要的时间。旋转延迟取决于磁盘转速，通常使用磁盘旋转一周所需时间的 1/2 表示。比如，7200rpm 的磁盘平均旋转延迟大约为  $60 \times 1000 / 7200 / 2 = 4.17\text{ms}$ ，而转速为 15000rpm 的磁盘的平均旋转延迟约为 2ms。

### 6) 数据传输时间（Transfer）

是指完成传输所请求的数据所需要的时间，它取决于数据传输率，其值为数据大小除以数据传输率。

硬盘有两个重要的性能参考指标：IOPS（每秒 I/O 数）和 Throughput（吞吐量）。IOPS 表示存储系统每秒内传输 I/O 的数量，Throughput 则表示每秒内数据的传输总量。二者在不同的情况下都能表示存储的性能状况，但应用场景不尽相同。同时，二者之间也存在联系。

- IOPS：用来表示 1s 内硬盘发起的连续 I/O 动作次数，其计算公式为  $\text{IOPS} = 1\text{s} / (\text{寻道时间} + \text{旋转延迟} + \text{数据传输时间})$

在通常情况下，广义的 IOPS 指的是服务器和存储系统处理的 I/O 数量。但是，由于在 I/O 传输过程中，数据包会被分割成多个块（Block），交由存储阵列缓存或者磁盘处理，所以对于磁盘来说这样的每个 Block 在存储系统内部也被视为一个 I/O，存储系统内部从缓存到磁盘的数据处理也会以 IOPS 作为计量的指标之一。

- 吞吐（Throughput）：用来计算每秒内在 I/O 流中传输的数据总量。这个指标在大多数的磁盘性能计算工具中都会被显示，在 Windows 文件复制时，就会显示 MB/s。通常情况下，Throughput 吞吐量只会计算 I/O 包中的数据部分，I/O 包头的数据则会被忽略在 Throughput 吞吐量的计算中。广义的 Throughput 吞吐量也被叫作“带宽”，用来衡量 I/O 流中的传输通道，比如 2/4/8Gbps Fibre Channel、60Mbps SCSI

等。但“带宽”会包括通道中所有数据的总传输量的最大值，而 Throughput 吞吐量则只包括传输的实际数据，二者还是有些区别的。

## 5.2.2 磁盘阵列

磁盘阵列（Redundant Arrays of Independent Disks, RAID）的基本思想就是把多个相对便宜的硬盘组合起来，形成一个硬盘阵列组，使性能达到甚至超过一个价格昂贵、容量巨大的硬盘。RAID 比单颗硬盘有以下好处：增强数据集成度、增强容错功能、增加处理量或容量。通常依照阵列的结构形式分成 RAID0、RAID1、RAID2、RAID3、RAID4、RAID5、RAID0+1 等类型。

RAID0 是最早出现的 RAID 模式，它将两个以上的磁盘串联起来，成为一个大容量的磁盘，在存放数据时，数据被分段后会分散存储在这些磁盘中。因为读写时都可以并行处理，所以在所有的级别中 RAID 0 的速度是最快的。但是 RAID 0 既没有冗余功能，也不具备容错能力，如果一个磁盘（物理）损坏，那么所有数据都会丢失。如图 5-4 所示。

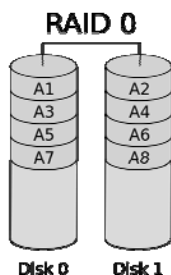


图 5-4 RAID0 数据组成结构

RAID1 叫作磁盘镜像，其原理为在主硬盘上存放数据的同时也在镜像硬盘上写同样的数据。若主硬盘（物理）损坏，则镜像硬盘会代替主硬盘的工作。因为有镜像硬盘做数据备份，所以 RAID1 的数据安全性在所有的 RAID 级别上是最好的。但无论用多少磁盘做 RAID1，也仅算一个磁盘的容量，是所有 RAID 中磁盘利用率最低的一个级别。如图 5-5 所示。

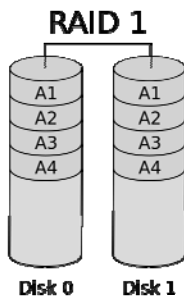


图 5-5 RAID1 数据组成结构

RAID 2~4 叫作校验模式，在磁盘组中有一个专门的磁盘用作校验，2~3 级别的差异主要体现在校验算法、数据传输方式上。如图 5-6、图 5-7 所示。

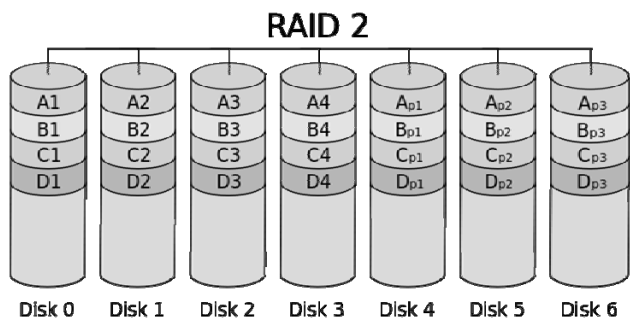


图 5-6 RAID2 数据组成结构

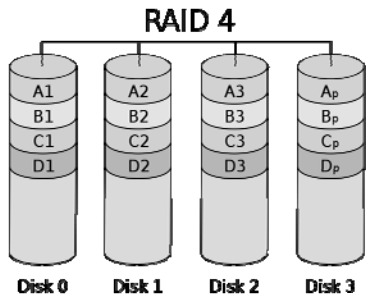


图 5-7 RAID4 数据组成结构

RAID5 为分布式校验模式，它的奇偶校验码存在于所有磁盘上，其中的 C<sub>p</sub> 代表第 C 带区的奇偶校验值，其他的保持一致。RAID5 的读出效率很高，写入效率一般，块式的集体访问效率不错。因为奇偶校验码在不同的磁盘上，所以提高了可靠性。但是它对数据传输的并行性解决并不好，而且控制器的设计也相当困难。RAID 3 与 RAID 5 相比，重要的区别在于 RAID 3 每进行一次数据传输，都需涉及所有的阵列盘。而对于 RAID 5 来说，大部分数据传输只对一块磁盘操作，可进行并行操作。在 RAID 5 中有“写损失”，即每一次写操作将产生四个实际的读、写操作，其中两次读旧的数据及奇偶信息，两次写新的数据及奇偶信息。如图 5-8 所示。

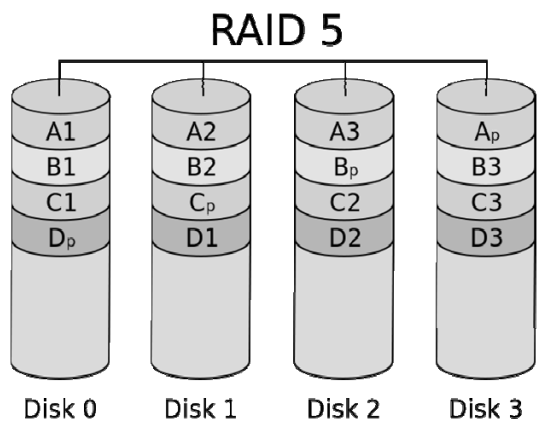


图 5-8 RAID 4 数据组成结构

RAID 10、RAID 01 是 RAID 0 和 RAID 1 标准结合的产物。RAID 10 是先做镜像，再进行条带式；RAID01 则与 RAID10 恰好相反，先分区，再将数据镜像到两组硬盘。如图 5-9 所示。

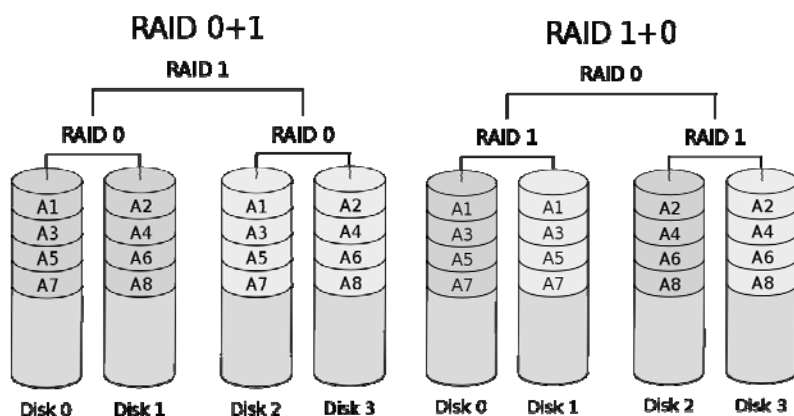


图 5-9 RAID 01、RAID 10 的数据组成结构

### 5.2.3 SCSI协议

在存储系统的世界中，SCSI（Small Computer System Interface）是主要的节点间的通信协议。它是一组专门用于计算机与其周边设备（硬盘、软盘、光驱、打印机等）进行数据交互的标准，不仅包含了通信协议，还涉及交互的命令、物理接口等规范，由于接入的硬件设备类型越来越多，接入网络拓扑也与具体的网络设备相关，所以 SCSI 逐渐演变成一套框架模型以兼容各种交互场景。如图 5-10 所示是 SCSI 框架，在这个体系中我们可以看到前端是以命令（Commands）的方式与设备交互，不同的设备硬件类型对应不同的命令。在后端，SCSI 通过协议分层有不同的通信方式（Interconnects），每一种通信方式对应不同的网络通信方式，包括直接通过总线相连的 SPI 接口、走存储网络的 FC 光纤、可运行在以太网上的 TCP/IP 等。

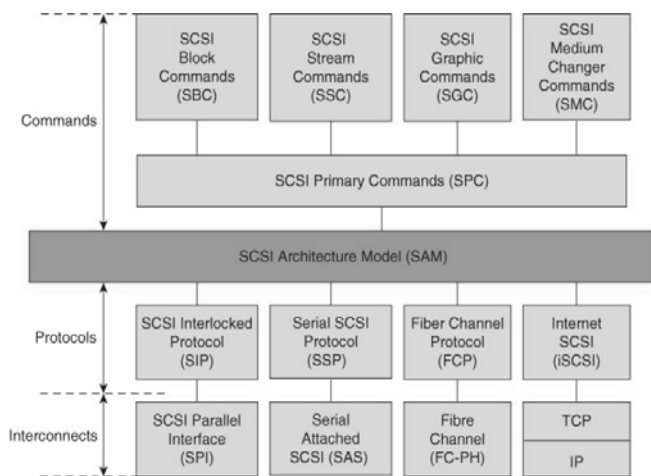


图 5-10 SCSI 框架

## 5.3 存储、计算分离

### 5.3.1 磁盘柜与盘阵

在计算机发展早期，存储系统位于计算机内部，磁盘直接连接到 I/O 总线的 SPI 接口上。但随着业务的发展，应用对存储的需求剧增，这时在计算机机箱内无法再将所有磁盘装下，于是将磁盘从服务器中分离，使用独立的机箱，在这个机箱中有独立的电源与散热系统，在内部的电路板上集成了 SCSI 线缆，所有磁盘接入到电路板上，外部有一个 SCSI 接口用来与服务器相连，这就是最初的存储、计算分离方式。

人们将这种简单的磁盘箱称为 JBOD (Just a Bound Of Disks)，由于其在机箱中没有放置 RAID 卡，所以在接入服务器后，操作系统将识别所有的磁盘，每个磁盘都是一个独立的块设备。这时可以使用软件层面的卷工具来管理 JBOD，例如 LVM (AIX、HP-UX、Linux)、DiskSuite (Solaris)、ZFS (Solaris)、Veritas Volume Manager (Unixes)。如图 5-11 所示。



图 5-11 JBOD 磁盘柜

在 JBOD 的基础上，如果在机箱内部加入 RAID 卡进行磁盘阵列控制，那么我们将这种自带 RAID 控制器的 JBOD 称为磁盘阵列或者盘阵。服务器通过 SCSI 看到的是一个或者多个块设备，具有一个或者多个 SCSI ID，所有逻辑磁盘都以 LUN 的形式在服务器端显示。

LUN (Logical Unit Number) 为逻辑单元号。在 SCSI 总线上可挂接的设备总数量有限，最多允许 16~32 个设备接入，每一个设备通过 Target ID (也称为 SCSI ID) 来描述，当磁盘阵列 (见图 5-12 所示) 中的设备数超过 SCSI 总线限制时，必须用一种扩展方式对设备进行描述，于是 LUN 的概念被引入，它扩充了 SCSI 的 Target ID，可以认为是 Target ID 的次级，在每个 Target 下可以有多个 LUN Device。



图 5-12 存储磁盘阵列



这时独立存储形成了两种重要的类型，一种是带有控制器的磁盘阵列，另一种是挂在大量磁盘上的 JBOD。

从服务器内部独立出来的磁盘阵列不断完善功能。在高可用性上，其采用主、备 RAID 控制器，通过心跳检测进行健康检查，在发现问题时快速进行切换。另外，随着对存储需求的不断提升，一个存储系统常常会拥有几十到几百 TB 的容量，这时采用大的主机系统来替换或者管理原来集成的芯片控制器，不仅可以提升处理性能，还可以在软件层面上引入多种功能。

由于 RAID 控制器的价格不菲，如果让多个磁盘柜共用一个控制器，则将大大降低成本，这也就形成了磁盘阵列与 JBOD 的组合，JBOD 连入磁盘阵列，磁盘阵列作为机头存在，而 JBOD 以磁盘柜的形式进行了存储扩展。

到目前为止，我们所讨论的存储、计算分离是点对点的，实际上存储区与服务器间的连接也可以是一张网络，我们将这种专门用来让服务器连接存储的网络称为存储区域网络（Storage Area Network，SAN）。存储区域网络在网络协议上是分层的，如果使用 FC（Fibre Channel）协议来承载 SCSI，那么也就形成了 FC 存储网络。

### 5.3.2 FC存储网络

存储、计算分离后，磁盘阵列在高可用、可扩展性上做了大量的完善，但由于在设备连接数量和通信距离上的限制，以及较差的在线设备扩展能力，SCSI 技术无法满足实际生产应用需求。而光纤技术在 SCSI 技术的基础上克服了 SCSI 的缺点，对底层的软、硬件全部更新、替换，使用光纤技术可以获得更快的速度、超大的容量、更好的安全性、更远的通信距离，连接设备数量也几乎无限多，支持在线增加和删除设备。

Fiber Channel（FC）技术标准是于 1994 年由 ANSI 标准化组织制定的一种适合千兆网数据传输通信的网络技术，其拓扑技术已被世界上所有的重要服务器及存储厂家所采纳，并成为下一代大容量、企业级数据存储的标准。伴随着网络存储，实时传输、视频点播和 CAD/CAM 等超大数据量传输应用出现了。

SAN（Storage Area Network）即存储网络是利用光纤通道在主机和存储系统之间形成的一个网络，这个网络突破了现有存储的架构及速度，消除了频宽的瓶颈，而网络与总线合二为一的特性使其不但和现有网络兼容并存，而且在数据的存储与传输上比 SCSI 更有拓展性、更易于使用，使服务器和存储系统的性能大幅提升。

光纤通道提供了以下三种不同的连接方式。

#### 1) 点对点（FC-P2P）

点对点允许两个 Node（一台服务器和一台存储设备）直接通信，如图 5-13 所示。这种

连接方式适用于对系统可用性及性能要求不高，但有一定的数据量、对速度有较高要求的应用场所。

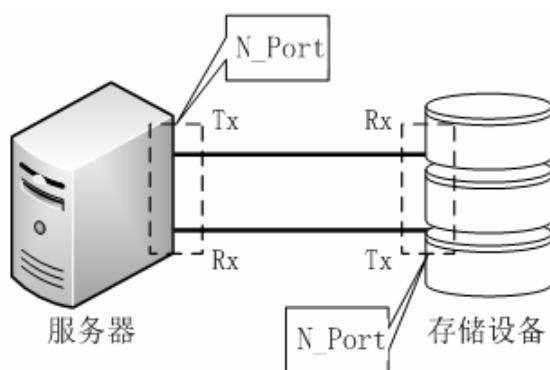


图 5-13 SAN 存储 FC-P2P 连接

## 2) 仲裁环 (FC-AL)

仲裁环是一个共享的环状网，其连接方式与 IBM 的令牌环网类似。在仲裁环拓扑中，设备必须根据仲裁访问环路，如图 5-14 所示。在实用性及可扩展性之外，对整个系统而言，其成本较低。

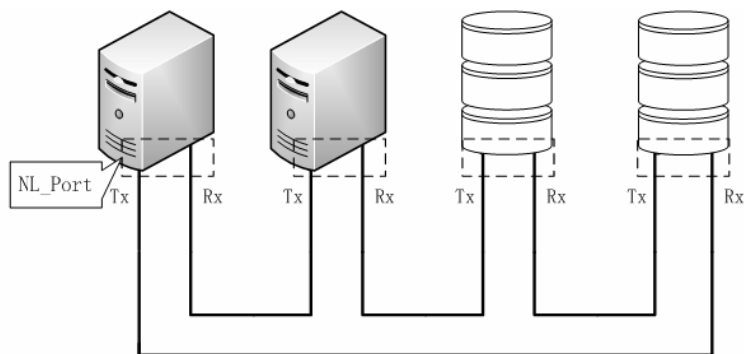


图 5-14 SAN 存储 FC-AL 连接

在仲裁环网络连接中，在逻辑上连入到网络中的所有设备首尾相连形成一个环状，但在物理上为了简化线路连接，一般会在中央放置一个光纤集线器，它负责连接所有的接入设备，在其内部将数据流汇成环，如图 5-15 所示。

## 3) 交换网 (Fabric)

交换网在主机和存储设备之间通过光纤通道交换机来连接，这种连接方式可形成有上千万台主机和存储设备的大型存储网络，交换网能为每个端口提供全带宽，其连接方式需使用专门的存储网络管理软件。如图 5-16 所示。

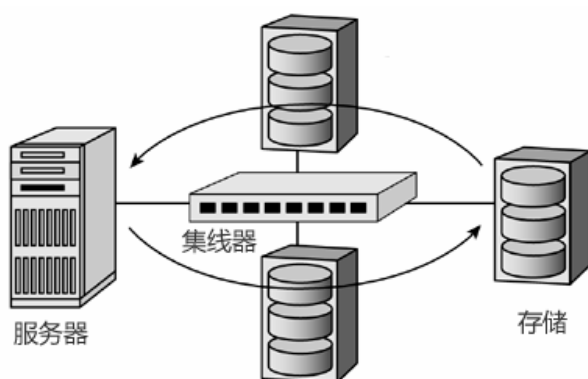


图 5-15 SAN 存储 FC-AL 物理接线

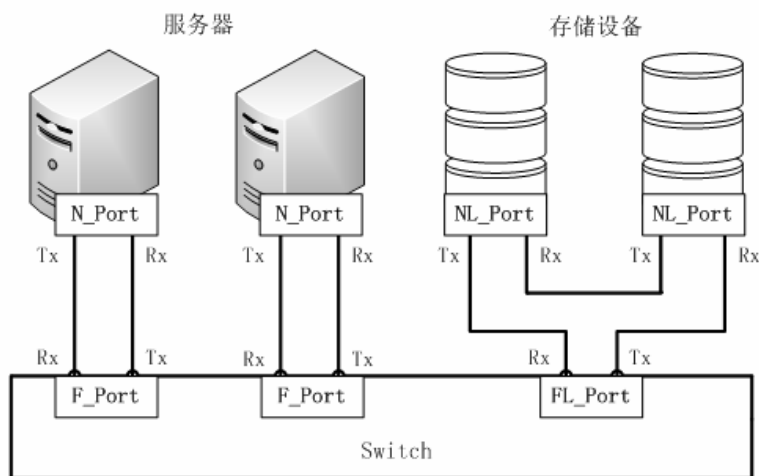


图 5-16 SAN 存储 FC-SW 连接

在 FC 网络中存在着 7 种类型的端口，分别是 N、F、L、NL、FL、E、G 端口。

N 和 F 出现在交换网拓扑中，如图 5-16 所示。N 是 Node，F 是 Fabric。N 代表一个严格的连接设备，它不执行任何基于逻辑块地址或文件信息的操作。F 是在光纤交换机中与 N 相连并为之提供服务的端口。

L 只存在于仲裁环拓扑中，如图 5-14 所示。L 是 Loop，在环中所有设备接入的端口都是 L。

当交换网与仲裁环以两种连接方式集成时，可以看作将仲裁环作为整体接入交换网中，在物理上则是将光纤交换机与光纤集线器互连。在这样的一个网络环境中出现了两种新的端口：NL 与 FL，如图 5-16 所示。FL 是与集线器相连的交换机端口，NL 是仲裁环接入到交换网的终端端口。

E 端口是在交换机之间互连的端口，因为交换机之间的互连，在设备之间需要做一些路

由选举的控制信息。

G 是通用端口或者说是“万能端口”，它可以是 F、FL 和 E 三种类型，其通过识别链路对端情况来自动配置。

### 5.3.3 FC协议栈

光纤通道网络与 TCP/IP 一样，有着自己的协议栈，它与 OSI 参考模型的对应关系如图 5-17 所示。

OSI 参考模型		FC协议栈
应用层		FC-4
表示层		
会话层		FC-2, FC-3
传输层		
网络层		
数据链路层		FC-2, FC-3
物理层		FC-1
		FC-0

图 5-17 FC 协议栈与 OSI 对应关系

- FC-0 层：描述物理接口，包括传送介质、发射机、接收机及其接口。FC-0 层规定了各种介质和与之有关的能以各种速率运行的驱动器和接收机。
- FC-1 层：描述了 8B/10B 的编码规则，该码型可以实现传送比特流的 DC 均衡，使控制字节与数据字节分离且可简化比特率，字节和字同步。另外，该编码具有检测某些传送和接收误差的机制。
- FC-2 层：信令协议层，它规定了需要传送成块数据的规则和机制。在协议层，FC-2 层是最复杂的一层，它提供不同类型的服务，分组、排序、检错、传送数据的分段重组，以及协调在不同容量的端口之间的通信所需要的注册服务。
- FC-3 层：为提供的一系列服务，是光纤通路节点的多个 N 端口所公用的。由于必要性限制，所以对这层尚未给出明确定义，但是它所提供的功能适用于整个体系结构未来的扩展。
- FC-4 层：提供了光纤通路到已存在的更上层协议的映射，这些协议包括 IP、SCSI 协议和 HIPPI 等。

### 5.3.4 FC寻址过程

如同以太网网卡的 MAC 地址一样，FC 网络中的每一个设备都有一个 WWNN（World Wide Node Name），其是一个 64 位的地址，由电器和电子工程协会（IEEE）标准委员会指定给制造商，在制造时被直接内置到设备中，是全球范围内的唯一地址。

FC 设备的每个端口都有一个 WWPN（World Wide Port Name），其如同 WWNN 一样，是世界范围内的唯一地址。

因为 WWPN 的地址太长，如果用它来进行寻址，则会影响路由性能，所以 FC 网络采用了一种映射方式，它在 WWPN 上做了一层映射，每个连入 FC 网络的接口都会被分配一个 Fabric ID，之后用这个 ID 而不是 WWPN 进行寻址、路由。这是一个 24 位的唯一地址，类似于 TCP/IP 中的 IP 地址，Fabric ID 由三个部分组成，每一部分占 8 位，分别是 Domain、Area、Port。

- **Domain:** 用来区分 FC 网络中的每一台交换机，在组件 FC 网络时，所有的交换机根据 WWNN 及选举相关参数选举出一台主交换机，然后主交换机负责为每一台交换机分配 Domain ID。
- **Area:** 用来区分同一交换机上的不同端口组。
- **Port:** 用来区分同一端口组中的不同端口。

我们可以计算出一个 SAN 网络最大的地址数目： $\text{Domain} \times \text{Area} \times \text{Ports} = 239 \times 256 \times 256 = 15\,663\,104$  个地址。

在 TCP/IP 网络中，第二层通过广播 ARP 来询问物理地址的所在，交换机仅做转发工作。而在 FC 网络中，交换机却智能得多，其内部运行着一个名称服务器进程，它本质上是一个对象数据库，当有设备连入进来时，交换机参与到设备的 Fabric ID 分配，同时将这个 ID 与 WWNN 的对应关系记录到对象数据库中。在接入设备向名称服务器注册之后，名称服务器会告诉接入的设备在当前网络上的相关节点与资源。

FC 网络中的交换机互相连接，它们之间运行动态路由协议，其中使用最普遍的是 SPF（最短路径优先）协议，通过路由协议交互网络中各节点的信息，从而进行节点间的无障碍通信。

如果接入 FC 网络中的设备知道了所有节点与资源的地址，同时 FC 交换机之间通过动态路由来选择最佳路径以保证数据通信，则这意味着设备可以访问任意资源。如果我们在名称服务器上设置，那么将某些资源地址隐去，但如果设备通过其他方式获取资源 ID，则它依然能够进行访问。这时在 FC 中引入了一个 ZONE 的概念，即分区，有一点类似于以太网交换机中的 VLAN 设置，只有在同一个分区内的节点设备才能互相通信，从而对资源进行保护。

### 5.3.5 FC交换机与适配器

FC 交换机通过光纤通道提供高带宽和低延迟的数据通信服务。在交换过程中，每一个端口都独享所有带宽，每一个连接都可以单独存在，与其他连接互不干扰。FC 交换机端口的数量为 8~96 口，甚至可以更多，其中包含智能交换硬件，使交换机所有端口中的任意两点可以建立连接。光纤交换机通过 E 端口（扩展端口）进行级联堆叠，这种方法可以使 FC 网络扩展到数千个节点，交换机堆叠最多可以达到 239 个。

FC 交换机在 SAN 存储架构中处于连接核心地位，它连接着主机和存储设备。当从一个设备发送一帧数据到交换机时，交换机在收到后，将该帧路由到适当目标设备中。实际上，一个帧可以在被完全接收之前就开始进行转发。FC 交换机很智能，它可以提供各种 Fabric 服务，包括在网络上定位其他节点的服务（简单名称服务），可以自动与 Fabric 中的其他交换机建立路由（动态路由），将设备分区（Zoning），还可以进行异常监视和处理等。

FC 交换机一般分为入门级、企业级和核心级。入门级交换机的应用主要集中在 8~24 个端口的小型工作组，适合低价格、很少需要扩展和管理的场合。它们往往被用来代替集线器，可以提供比集线器更高的带宽和提供更可靠的连接。Brocade 6505 是一款有 24 个端口的 1U 位入门级交换机，最高可提供 16 Gbps 的速度。如图 5-18 所示。



图 5-18 入门级 Brocade 6505 交换机

企业级交换机提供比入门级交换机更高的可靠性和更强大的性能，更多地用于连接入门级交换机。核心级交换机一般位于大型 SAN 的中心，使若干边缘交换机相互连接，形成一个具有上百个端口的 SAN 网络。核心交换机的其他功能还包括：支持光纤以外的协议（例如 InfiniBand）、支持 2Gbps 光纤通道、高级光纤服务等例如安全性、中继线和帧过滤等。如图 5-19 所示。

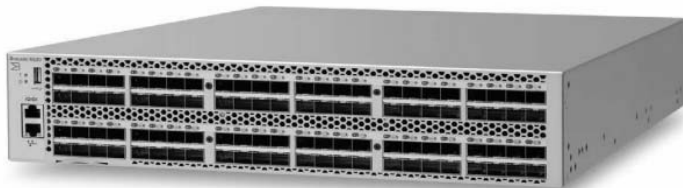


图 5-19 企业级 BROCADE 6520 交换机

最后想要连入到 FC 网络中的服务器需要配备 FC 适配器，或者叫作 FC 主机总线适配

器，即 FC HBA（Host Bus Adapter）。HBA 可以为包括 PCI 和 SBUS 在内的多种内部总线提供相应接口。插入 HBA 卡后，在操作系统上需安装相应的 HBA 卡驱动程序。如图 5-20 所示。

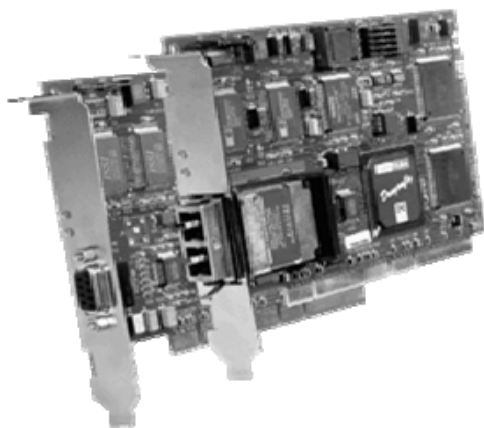


图 5-20 FC HBA

### 5.3.6 FCoE与iSCSI

协议分层的最大魅力在于通过组合不同协议，让相同的应用功能运行在不同硬件设备、网络环境中。数据存储功能就是这样的一个例子，使用 FC 网络来承载 SCSI，让 I/O 存储从服务器中独立出来。

以太网在数据中心无处不在，如果能够将 FC 协议融合到以太网架构中，那么这将简化网络拓扑环境，降低布线成本，而协议分层让这一切变为可能。

FCoE 采用增强型以太网作为物理网络传输架构，能够提供标准的光纤通道有效内容载荷，能够像标准的光纤通道那样为上层软件层（包括操作系统、应用程序和管理工具）提供服务。FCoE 仍然保留了 FC 中 N\_Port、F\_Port、E\_Port 的结构，以及 FC 的管理模式，在上层处理方面和 FC 网络没有区别。而在底层协议上，其直接在以太网上承载，属于纯二层网络架构，数据传输不使用 IP 网络，从而避免了 TCP/IP 开销，如图 5-21 所示是 FC 协议栈与 FCoE 协议栈的对比。

和标准的光纤通道 FC 一样，FCoE 协议也要求底层的物理传输是无损失的。因此，国际标准化组织开发了针对以太网标准的扩展协议簇，尤其是针对无损 10GB/s 以太网的速度和数据中心架构。这些扩展协议簇可以进行所有类型的传输。这些针对以太网标准的扩展协议簇被国际标准组织称为“融合型增强以太网（CEE）”。FCoE 技术有以下优点：光纤存储和以太网共享同一个端口；更少的线缆和适配器；软件配置 I/O；与现有的 SAN 环境可以互操作。如图 5-21 所示。

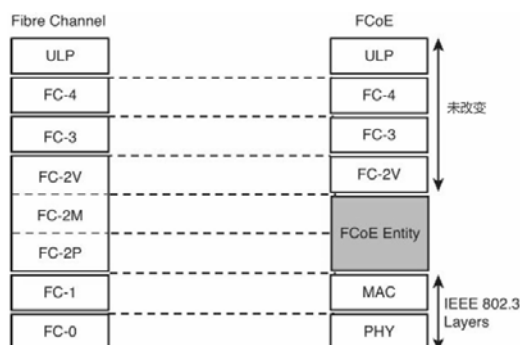


图 5-21 FC 协议栈和 FCoE 协议栈的对比

iSCSI (Internet Small Computer System Interface) 为 Internet 小型计算机系统接口，又称 IP-SAN，是一种基于 TCP/IP 及 SCSI-3 协议的存储技术，由 IETF 提出，并于 2003 年 2 月 11 日成为正式标准。与传统的 SCSI 技术相比较，iSCSI 技术有如下三种革命性的变化：

- 把原来只用于本机的 SCSI 协议通过 TCP/IP 网络承载，使连接距离可以无限延伸；
- 连接的服务器数量无限（原来的 SCSI-3 的上限是 15）；
- 由于是服务器架构，因此也可以实现在线扩容以及动态部署。

虽然 iSCSI 和 FCoE 都运行在以太网上，但 iSCSI 并不需要像 FCoE 一样要求物理传输无损失，它可以跑在与企业网络架构完全兼容的以太网上，并且通过 IP 进行路由。iSCSI 常常被认为是 FC 的一种低成本替代方案。

iSCSI 将存储设备端，通过 iSCSI 的 target 功能实现可以提供块设备的服务器，再通过 iSCSI initiator 功能实现能够挂载 iSCSI target 的客户端，如此便能通过 iSCSI 协议来进行对块设备的磁盘管理。iSCSI 架构主要将存储与使用的主机分为两部分，分别如下。

- iSCSI target: 是存储设备端，负责管理 JBOD 或磁盘阵列设备，也能够将一台普通的 Linux 服务器模拟成 iSCSI target，通过 TCP/IP 对外提供块设备服务。
- iSCSI initiator: 是使用 iSCSI target 的客户端，也就是说，想要连接到 iSCSI target 的用户，则必须要安装 iSCSI initiator 客户端程序后才能使用 iSCSI target 提供的块设备存储。

## 5.4 存储访问类型

### 5.4.1 NAS与SAN

在之前的章节中我们一直围绕着如何将 SCSI 协议服务接口提供给计算机服务器进行讨



论，在这个层面上服务器看到的都是块设备，而存储设备可以在服务器内部直连 I/O 总线，也可以拆分成独立的磁盘阵列柜，还可以基于 FC 构建存储区域网络，即 SAN。

在 5.1.2 节讨论过文件存储的三个层级，在第三层级上，操作系统对所分配的逻辑卷进行文件系统格式化，之后用户进程才能真正地通过 VFS 虚拟文件系统接口读写数据。也就是说，最终用户需要的是文件系统，而不是有待格式化的块设备。这也就是 NAS 与 SAN 的最大区别所在，NAS 是面向文件的网络存储。

块（Block）是最直接的数据访问方式，用来表示存储设备上数据的最小存储单元，块由一组固定长度的字节序列组成，例如其表示磁盘的一个扇区，大小为 512Bytes。服务器如果通过块方式访问存储设备，则拥有了对磁盘“最大”的控制权限，基于文件、对象及记录的访问方式都是建立在块之上的。

面向文件（File）的数据存储类型是符合人类习惯的一种方式，如同在日常生活中管理文件，将文件整理好放入抽屉。前面已阐述过在计算机世界里一切都是二进制，文件除了用户关注的有效数据，还有一部分用以描述文件自身的元数据，例如文件名、大小、修改日期、所有者等。每个操作系统都提供了一套文件管理接口。

NAS（Network Attached Storage）是一种面向文件访问的计算机数据存储服务，它在网络中向异构的客户端提供服务。我们可以认为 NAS 在一台服务器上通过 SCSI 协议连接到块设备后，将设备磁盘进行文件系统格式化，之后通过网络将这个卷共享给不同的客户端使用。它在网络中使用的文件共享协议包括 NFS、SMB / CIFS 或 AFP。

基于远程的文件共享协议主要有以下两大类。

### 1) NFS（Network File System）

NFS 是当前主流异构平台的共享文件系统之一，主要应用在 UNIX 环境下，最早由 SUN microsystem 开发，能够支持在不同类型的系统之间通过网络进行文件共享，被广泛应用于 FreeBSD、SCO、Solaris 等异构操作系统平台，允许一个系统在网络上与他人共享目录和文件。NFS 传输协议用于服务器和客户机之间的文件访问和共享通信，从而使客户机远程访问保存在存储设备上的数据。

### 2) CIFS（Common Internet File System）

Microsoft 推出 SMB（Server Message Block）后，又将其扩展到 Internet 上，成为 CIFS。CIFS 采用 C/S 模式，基于网络协议 TCP/IP 和 IPX/SPX，通过 NTLM 和 Kerberos 在客户端与服务器之间提供 AD 认证。

NAS 与 SAN 并不冲突，可以说它们完全是两类事物，SAN 强调的是提供面向块设备的存储网络，而 NAS 强调的是提供面向文件访问的网络存储。NAS 还可以与 SAN 进行结合，之前说过，在存储设备的机头上是一个服务器的 RAID 控制器，如果在服务器上增加 NAS 功能，进行文件系统格式化，并对外提供 NFS 等网络文件协议的服务，那么这个控制

器马上就变成了一个 NAS 机头。如图 5-22 所示是 NAS 和 SAN 的关系图。

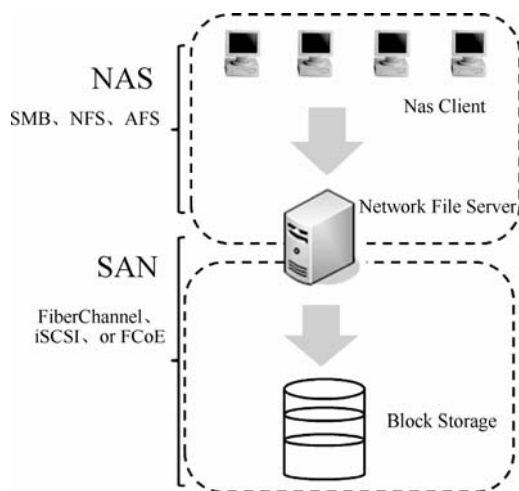


图 5-22 NAS 和 SAN 的关系图

## 5.4.2 分布式存储

前面对存储的讨论可以认为是很传统的，从一台单机服务器访问本地磁盘，到将磁盘分离到外部，形成独立磁盘阵列机柜，再到通过 FC 网络形成 SAN 网络，最后在各种块设备服务之前实现网络文件系统 NAS。随着互联网的兴起，云计算、大数据成为 IT 的两大热门领域，在这两个热词的背后是对基础资源的海量需求，其特点是使用规模大，成本要求低，在这样的背景下出现了分布式存储。

在成本控制方面，分布式存储采用大量的普通 PC 服务器作为存储源，将数据分散存储到多台网络独立的设备上；而传统存储则采用专门的磁盘阵列机柜，集中式存放数据。在使用规模方面，分布式存储必须考虑到其扩展性，保证可动态地构建几百台甚至几千台集群规模。组成分布式存储的 PC 服务器底层仍旧是块设备存储。分布式存储系统在传统服务器接口上重新包装了一层，然后实现不同的文件系统接口对外提供服务，解决扩展性问题的层级自然放在了文件系统及通信协议层面，准确地说，应该称之为分布式文件系统。

不同的文件系统接口对应于不同的应用场景。分布式存储流行以下三种接口。

- 对象存储：键值接口的存储系统，其接口是简单的 Restful HTTP API，Swift、S3 等都提供这种接口服务。每个对象就是一个文件，通过唯一的 ID 进行标识。每次操作都是针对整个文件的，包括 GET、PUT、DEL 等，每次只能全写或全读文件，要求具备足够的网络 I/O 带宽。这种存储适应于跨互联网的附件存储、传输，有时甚至作为 CDN 加速使用。
- 文件存储：文件存储用于保持与 POSIX 接口的兼容性，在这种支持下，在遗留程

序中对文件的操作可以直接迁移到分布式存储上来，跟传统的文件系统接口如 ext4 等没有区别。NAS 的 NFS、CIFS 协议等都属于文件存储类型。

- **块存储**：这种接口通常以 QEMU Driver 或者 Kernel Module 的方式存在，需要实现 Linux 的 Block Device 接口或 QEMU 提供的 Block Driver 接口。我们知道块设备在文件存储系统的最底层，如果在分布式场景中提供块存储，那么其在原有的“真实”块设备上又覆盖了一层“虚拟”块设备接口。块设备通常使用在对 I/O 延迟敏感的环境中，例如数据库 I/O 访问、虚拟机存储等，这种场景不适合异地部署分布式存储，必须放在与主机靠近的地方，以保证 IOPS。

除了上面所说的三种接口，还有列式存储、文档型存储、消息队列存储、HDFS 分布式文件系统存储等。按照这种方式划分，分布式存储分类将无穷无尽，实际上我们可以将其看作在原有单机块设备或文件系统上包裹了一层新的访问接口。但是有一点需要注意，接口会影响到存储的数据结构，数据结构又会影响到存储的检索算法。

在分布式存储中，一份数据会做多份副本以保存在不同的节点之中，从而实现数据备份，以及服务的高可用。分布式领域有着著名的 CAP 理论，如下所述。

- **Consistency（一致性）**：数据一致更新，所有数据变动都是同步的，客户端在所有数据节点上看到的内容都是一样的。
- **Availability（可用性）**：有好的响应性能，数据节点在任何时候都可以对外提供服务。
- **Partition tolerance（分区容错性）**：有可靠性，允许数据节点异常或者数据节点之间通信异常。

定理：任何分布式系统只能同时满足以上两者，没法三者兼顾。

如图 5-23 所示是一个分布式存储场景，假设有  $1 \sim n$  个节点，其中的数据会进行副本同步，如果我们要求在所有节点看到的数据一致（图 5-23 中片段 2 为数据），也就是获得了 C（一致性），同时要求所有节点都能对外提供服务，也就是获得了 A（可用性），那么在 CA 的基础上，我们必须保证所有节点之间的网络正常，这样才能有效地进行数据同步，即丢失了 P（分区容错性，允许通信异常）。

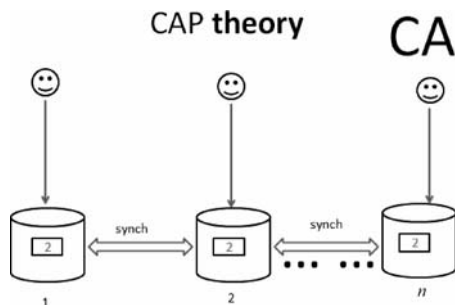


图 5-23 CAP 理论中的 CA

如图 5-24 所示是 CAP 中的 AP，在这种场景下允许网络通信异常，即拥有 P，造成的结果是某些节点无法有效同步数据，例如节点 N。但又必须保证所有节点可以提供服务，这又要求 A，可想而知，数据的一致性即 C，是无法保持的，从图 5-24 可看到  $n$  节点数据与其他节点并不一致。

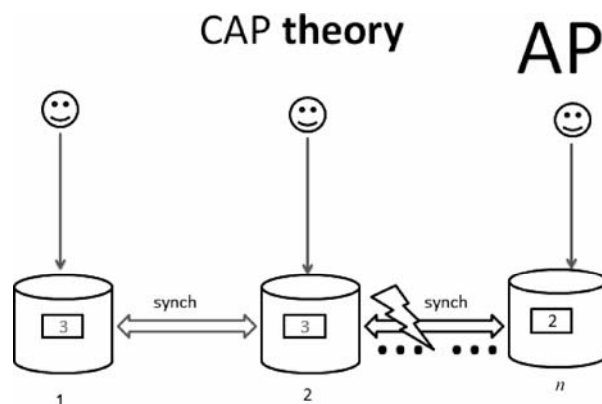


图 5-24 CAP 理论中的 AP

图 5-25 的情况与图 5-24 类似，但为了保证数据的一致性 C，从而将数据不一致的服务进行隔离，并且  $n$  节点不再对外提供服务，这就是丢弃了 A 即可靠性，剩下 CP。

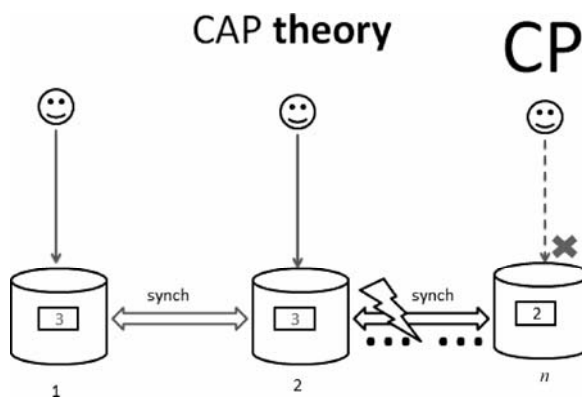


图 5-25 CAP 理论中的 CP

CAP 对架构师的忠告是不要将精力浪费在如何设计能满足三者的完美分布式系统上，应该进行取舍。



## 第三部分 平台实现

在建立起 PaaS 概念模型，回顾计算、存储、网络三大资源之后，这一部分让我们采用主流的开源产品来实现一个完整的 PaaS 平台，包括资源管理、任务调度、计算单元打包、分布式协调、日志集中等。通过学习本部分内容，我们将看到一个可扩展、自定义的开放 PaaS 平台。

# 平台功能与架构

我们需要在三大基础资源之上实现 PaaS 的概念。PaaS 是一个庞大的体系结构，私有 PaaS 平台需要适应多种基础资源组合的形式，不同的网络拓扑、不同的服务器及存储设备要能够向下兼容，保持足够的灵活性。对三大资源的日常管理就是运维，将运维需求进行整理，将这部分工作标准化、自动化，才能对上提供服务自助 API 接口，结合配置管理与服务管理构建一个完备的 PaaS 平台。PaaS 是技术实现与管理规约的双层结合。

运维工作之下的三大基础资源无论以何种形态出现，如不同的网络拓扑、服务器类型、存储接口等，无论这种形态要支持多大的应用场景，如 Vxvlan 网络、高端存储等，有一点非常重要并且需要在此基础上保证：资源提供方式要遵循一定的标准，没有标准，其上的运维工作则无法实现全面自动化，即无法再向上层 PaaS 提供用户自助接口。基础资源的标准其实又是另外一层接口，它要将三大基础资源的复杂性掩盖在这个标准接口之下。这个标准应当包括网段的划分、防火墙策略规则、服务器版本、域名配置规则等，对于运维工作而言，它能够依据这些规则将工作自动化。其重点不在于如何构建简单可标准化的基础资源设施，而在于中间层的运维需求自助化和运维管理准则。在整个过程中需要底层资源标准化、运维管理自动化、用户服务自助化的层层推进来构建最终的私有 PaaS。

## 6.1 平台运维需求

一个完整的分布式 PaaS 平台需要实现底层 IT 运维服务的自动化，常规的运维工作主要包括四个方面：软件配置、服务部署、服务发现、监控恢复。

### 6.1.1 软件配置

软件配置要回答的问题是在哪里安装软件、修改哪些配置文件、应用逻辑如何放置。这个动作建立在 OS 资源已分配的条件下，囊括了软件安装、配置甚至应用程序发布。

### 1. 配置管理工具

对于标准化软件大规模安装的场景，开源配置管理能够给运维人员带来很大的帮助。但其也有无法满足的场景，很多软件的配置修改并不基于文件编辑，而是通过 UI 界面操作的，这种情况下的配置管理工具无法实现自动化。另外，软件是介质，是静态的，实例以动态的进程形式运行，如何基于一份介质运行多个实例是配置管理工具无法解决的问题。例如，按照标准方式安装了一个 Tomcat，如果需要基于一份介质运行启动多个 Tomcat 进程，就需要运维管理人员非常熟悉 Tomcat 的配置关系，而配置管理工具则无法解决这个问题。

### 2. 虚拟机镜像克隆

通过虚拟机的镜像克隆是一种简单有效的方法，但其缺点也非常明显，对计算资源和存储资源都会造成极大的浪费。

## 6.1.2 服务部署

应用逻辑文件的同步归为软件配置部分，在这里服务部署要解决的问题是在哪里运行服务，如何运行服务，这不仅依托于底层操作系统资源，还需要将应用逻辑、依赖的容器、程序包提前准备好，并可在分布式环境中快速复制与流动。

## 6.1.3 服务发现

服务发现指在服务启动以后，关联方（包括终端用户与关联系统）如何访问到它。在分布式系统中有一个全局可靠的区域记录服务的地址信息，客户从这里获取信息进行访问，全局区域可能是简单的 DNS 服务器，也可能是具备消息通知功能的分布式协调系统。

## 6.1.4 监控恢复

在服务能够自动配置、部署与发现的最后环节，运维需要对服务本身进行保护，这就需要监控管理与容灾恢复来保障服务稳定运行。监控管理与容灾恢复与以上三点不一样的是它独立构成一个系统，并不参与到服务的构建过程中。

## 6.2 平台功能划分

为了能够实现 PaaS 平台，我们需要保证运维的 4 个主要工作内容实现自动化，下面这些功能全都是围绕着这个目标而引入的。

### 1. 计算单元打包

虚拟机镜像、配置管理工具所负责的工作就是将应用逻辑计算单元进行打包。计算单元包含了运行业务系统的全栈组件，其涵盖了操作系统、中间件、依赖包、业务逻辑程序等。在分布式平台中我们选择 Docker 作为一个轻量级容器，它比虚拟机更加节约资源，同时可以基于一份软件介质运行多个实例。Docker 也有其缺点，例如不支持 32 位平台，不支持 Windows 服务器。

### 2. 资源动态分配

与云计算的 IaaS 不同，用户并不关注如何获得 CPU、内存、存储资源，仅关注应用计算逻辑的运行，希望资源是动态分配、弹性扩容的。在分布式平台中需要一个统一的资源管理者，它将平台内的所有资源抽象成一个整体，如同一个巨型操作系统，按用户的需求动态地分配资源。

### 3. 作业调度功能

作业调度器与资源管理器的最大不同在于其要对运行中的应用服务负责，包括启动、停止、监控服务，以及在服务失效时将故障转移。在最初的分布式架构设计中，人们常常模糊了作业调度与资源管理二者之间的界线：分布式平台是为某一专属计算类型服务的，例如 Hadoop 平台为 MapReduce 计算类型服务；作业调度与资源管理的交互频度高，合二为一后的效率更高。但随后人们发现资源管理器的功能是相对稳定的，而作业调度器因为任务类型多样而易变，并行计算有 MapReduce、Stream，普通计算有 Service、批处理等，每一种计算类型的作业调度方式完全不同，如果将资源管理器与作业调度器绑在一起，则会失去分布式平台的计算灵活性。

### 4. 分布式存储

PaaS 平台的主要功能是实现计算、存储、网络资源的动态分配，网络用于任务之间的通信，更多地表现为开放与隔离，以及防火墙策略的控制。计算资源的衡量指标与时间关系，以及单位时间内完成的指令数量总是受 CPU 主频的影响而无法将所有的计算资源抽象成一个有无限能力的单核 CPU。对 CPU 资源的分配实际上还是将原始任务拆分成可以并行处理的子任务来提升的，Web 服务的用户请求、大数据的 MapReduce 分析、多线程、进程并发等都属于这种范畴。而存储资源则不然，存储的分配与空间有关系，在分布式系统中我们使用分散在各服务器上的存储资源，如果没有一种方式将这些分散点聚合成有一个逻辑的可不断扩展的大存储，那么存储资源的分配将变成一种受限的方式。当一个应用的生命周期趋于成熟时，其用户访问量剧增，随之而来的是存储需求的持续增长，这时如果由于物理存储节点之间的分割而无法对上层应用提供连续性资源，那么势必要求应用本身调整程序存储逻辑。这个问题应当在分布式平台的底层解决，对上提供面向块、文件及对象的存储结构，同时做到可扩展性及可伸缩性。



### 5. 分布式协调系统

在单机的多线程并发任务中，对全局变量的保护显得尤为重要，必须有一种方式将全局变量的一组操作变成原子性的，即在这个过程中这组操作不会被另一个任务所修改。在 Java 语言中会使用 `synchronized` 原语对内存段加锁进行保护。同样的同步控制在数据库中以事务的形式出现，在关系型数据库中定义的事务隔离级别无非是所共享的全局变量的范围大小，即需要锁定的是一行记录、一个表还是整个数据库。数据库的同步采用了优雅的多版本控制，在所有操作完毕后进入提交阶段来判断所修改数据的版本。在分布式系统中同样存在并发任务对全局变量访问的一种场景，同样需要进行同步控制。分布式协调系统模型有中心化、令牌环、组播通信等模型，在我们的分布式平台中需要构建此功能模块。

### 6. 数据共享中心

在分布式平台中需要有一个数据共享中心来放置应用之间交互的数据，满足应用之间数据的快速交互及互相访问，同时不会让应用耦合在一起。共享中心依据数据类型不同而不同，可以是一个 DNS 服务器，也可以是分布式消息队列。

### 7. 日志集中管理

在一般情况下日志以文件的形式存放在本地操作系统上以供查询，而在分布式系统中，计算单元不会再固定于一个物理节点上。如果日志仍以文件形式存放在本地，那么随着计算单元的漂移，日志将留存在与计算单元没有关系的物理节点上，对于系统管理人员、运营人员来说日志查询检索将变成一门繁杂的工作。在分布式平台上构建集中的日志管理平台，将各种类型的日志收集、索引好，以消息的形式看待每一条日志将成为分布式平台的一个重要功能。

### 8. 监控巡检管理

监控巡检管理实际上是一个很重要而且代价很昂贵的独立系统，它并不嵌入在整个运维流程中，与其他基础资源相关联。监控的复杂在于它比配置管理的数量级更大，要对每一个配置项所包含的监控项进行监控，例如对于一个服务器配置项，我们需要监控内存、CPU、磁盘、日志等。另外监控对象的种类很多，操作系统又涉及多种平台，JVM 有多种类型，网络设备的 SNMP 有特殊的 OID 等。这里仅讨论监控的一般管理方法及一些通用实现的举例。

### 9. 软件自动配置

尽管在分布式平台中采用了计算单元打包的方式来将计算逻辑快速在分布式节点中进行漂移，而这种计算逻辑“漂移”的方式属于新软件配置方法。“上一代”软件配置工具采用命令批量下发与软件状态管理的方式完成。在分布式平台中依然需要这种批量命令下发的工具，对于运行状态计算节点的批量控制需求在分布式场景中依然存在。

## 10. 平台门户与 API

将以上资源聚合在一起后，需要一个统一的接口对外提供服务。我们将其分为两类：面向用户的 Web 门户和基于 JSON 格式的 API 服务器。门户要实现用户管理、权限管理及配置管理功能，而基于 API 的服务器关注于将以上资源聚合成一个完整服务，对外提供可编程的接口。

## 6.3 平台高阶架构

为了满足以上功能，我们需要的 PaaS 组件高阶架构如图 6-1 所示。

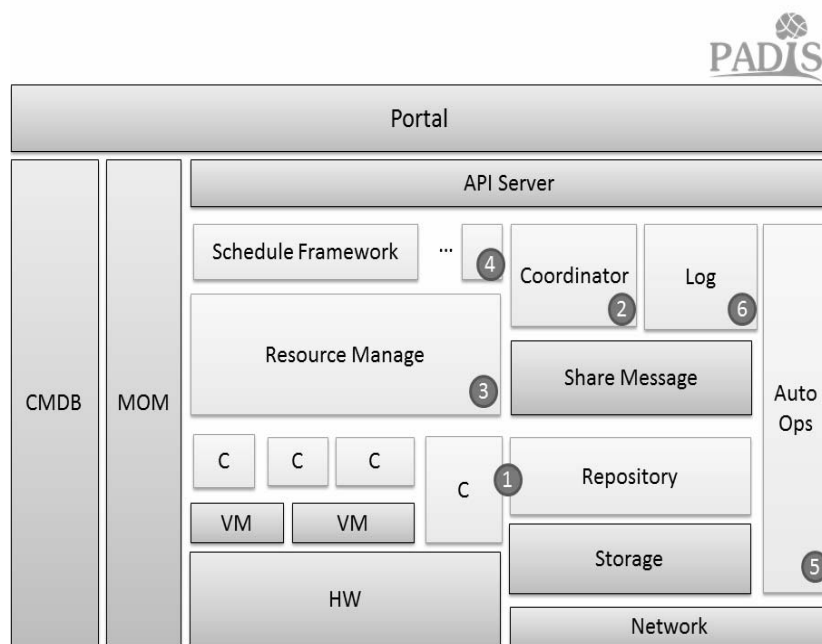


图 6-1 PaaS 组件高阶架构

- **Portal:** 整个 PaaS 平台有自己的 Portal 界面，整合后续所有的资源，面向用户提供服务。
- **API Server:** Portal 与底层资源之间有一层 API Server，它提供 Restful API，将底层所用资源的管理功能做封装，可以同时向前端的多种来源需求提供服务，其中 Portal 是主要的服务请求来源。
- **ComputeUnit:** 采用 Docker 容器作为计算单元打包的利器，随后可以看到将完整的应用栈打包在一个镜像内，类似于 Java 语言所宣称的那样，实现一次打包，到处

使用，而 Docker 所囊括的范围更加广泛。在 Repository 上使用 Docker 的仓库来保存所有的计算逻辑单元。Docker 容器默认通过端口转换向外提供服务，为了使每一个容器都能够像一个虚拟机样，我们要使用 pipewalk 之类的工具，从 IP 地址池中获取一个地址附加到活动的容器上，这个动作可以在启动一个容器后作为钩子程序实现。

- **ResourceManage:** 在分布式平台中有一个资源管理器，它将数据中心的所有资源抽象成一个“大操作系统”，资源管理器并不向用户应用程序直接提供接口，而是将资源分配给不同的调度框架，这些调度框架可以用于长任务型服务、批处理、后台任务，以及流程计算等各种类型的资源需求应用。资源管理的开源组件的最佳选择有两个，一个是 Hadoop 生态圈的 YARN，另一个是伯克利大学贡献的 Mesos，本书选择了轻量、灵活的 Mesos。
- **ScheduleFramework:** 调度器与资源管理是紧密相连的，在 Hadoop 1.0 中这两个功能放在了同一模块中实现，资源管理侧重于对三大资源的分配算法，而调度器要面向上层多变的应用类型，将稳定与多变的功能放在一起，既丢失了资源管理的灵活性，也加重了工作负载。在 Mesos 之上有各种可选调度框架，包括长任务的 Marathon，它结合 Docker 实现分布式平台上 Service 类型的计算逻辑动态伸缩与漂移。定时任务的 Chronos 能够将企业内部的 ETL 任务统一到平台中，共享计算资源。最后在 Mesos 上可构建 Hadoop、Spark 等大数据计算框架。
- **Coordinator:** 在分布式平台的中心有一个类似于交通枢纽的协调管理系统，用于分布式协调锁，同步各类组件状态。ZooKeeper 是开源社区中成熟的分布式协调系统，私有 PaaS 的资源管理、调度框架都会使用 ZooKeeper，甚至主流的消息队列 Kafka 在分布式场景下也会与 ZooKeeper 集成。
- **AutoOps:** 我们可以看到 AutoOps 自动化运维的常用工具，满足动态资源的批量命令与静态软件配置需求，在这里我们采用了 Saltstack 作为使用 Docker 计算单元打包的一个补充。
- **Log:** 在讲 12-factor 时强调过，日志不是文件，日志是一条条消息，在最终的 PaaS 计算单元中，本地将不存储任何日志，而是发送到远端的集中管理处。在日志集中管理部分采用了开源社区流行的 ELK 组件，我们会看到如何将所有节点的日志导入一个集中部分进行查看与展示。
- **其他:** 在存储方面，大型企业会采用专用存储设备保存核心数据，对于核心的交易请求保证 I/O 响应时间与吞吐。在 PaaS 中，存储场景与这类企业级应用核心交易存在区别，更多的是需要一个接口简单、成本低廉的分布式存储，通过应用上的多级缓存缩短 I/O 响应时间。当前流行的开源分布式文件系统有专用于大数据的 Hadoop HDFS，以及提供多种存储接口类型的 Ceph。HDFS 与 Ceph 可以结合使

用，Ceph 在最近推出的 HDFS-plugin 亦趋于成熟。另外在基础设施一致的情况下，使用 NUSE（Network Stack in Userspace）可进一步提升分布式存储性能。

对于长任务型的 Web 服务，我们需要增加负载均衡。负载均衡的硬件设备，例如 F5，可以通过其提供的 API 集成到 PaaS 平台。另一种选择是基于软件的 Haproxy。在一般情况下 Haproxy 与 Keepalived 这两个组件要一起使用，前者用于负载均衡功能，后者负责切换虚拟服务 IP 地址，实现高可用功能。在 PaaS 环境下，由于 Haproxy 所在的操作系统实际上也是一个容器，所以 PaaS 的调度框架会对 Haproxy 进行监控，一旦服务不可用，则将立即启动另一个 Haproxy 容器来满足高可用需求，因此 Keepalived 的心跳检测切换对 PaaS 来说没有太多的必要性。

## 6.4 企业应用迁移

### 6.4.1 企业应用很“厚重”

我们在第 1 章中学习了 12-Factor 规范，它可以让应用在 PaaS 平台上发挥最大的效用。但有一个不得不面对的问题，就是现有环境存在着大量的遗留系统，这些系统原本是企业级应用，在要迁移到 PaaS 上时，我们必须考虑提前对应用进行改造。

企业应用无法迅速向 PaaS 迁移的原因在于企业部署架构、应用架构等都背负着“厚重”的历史包袱，不管是核心交易还是普通查询，这些功能模块都不加区分地嵌套在“厚重盔甲”内，以此保证系统的安全性。例如某些企业将系统放在无标准的 NAT 双向防火墙内，这就很难通过自动化手段来开通策略，实现应用的自动扩容与伸缩。还有一种情况，在系统内部有一个独立的密码库，只有向密码库注册了的主机，在经过授权后才能对密码库中的某些内容进行访问并获取密码，这里的密码很可能仅仅是为了连接数据库。还有些情况是某些商用中间件有中心化的控制台，通信方式采用组播通信等，这些情况直接违反了 12-Factor 规范。

企业级应用遗留系统被层层枷锁套住，应用架构、安全架构都锁定了它，导致其无法迁移 PaaS，无法实现快速扩容、复制。这类应用可以上 PaaS，却体现不了 PaaS 的最大价值。

### 6.4.2 应用部署架构

大多数企业应用部署架构采用三层模型：Web 服务器、应用服务器与数据库。如图 6-2 所示。

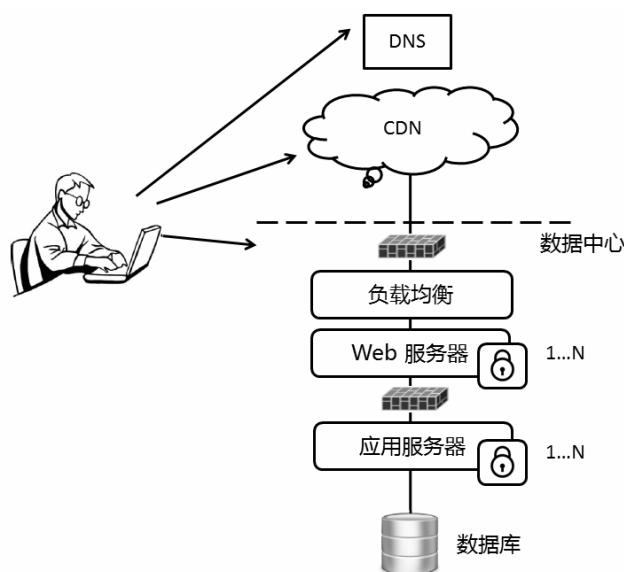


图 6-2 应用部署架构

Web 服务器负责接收用户输入、提供静态资源访问，以及与外部 CDN 加速关联。应用服务器采用商业的中间件，将应用逻辑包含在其中，向应用提供事务、数据访问等多种中间件服务。最后所有的数据持久化都在数据库中完成，该层一般采用商用的关系型数据库。Web 服务器、应用服务器在访问量不大的情况下，并不会引入缓存，所有的状态会话都保存在当前实例中。

Web 前端会采用硬件负载均衡设备。它不仅仅将请求转发到后端多个 Web，还涉及健康检查、重定向规则、SSL/TLS 证书、内容压缩等一系列功能。在这个节点上也会采用专用的商用设备。

用户通过 DNS 来查询域名所对应的 IP 地址。大型企业都会有自己的 DNS 服务器，并负责自己公司的专属 Zone，上一级域名商将对应的 NS 地址发送给企业 DNS 服务器。应用域名一般与 IP 地址一一对应，如果有多个服务出口提供服务，那么将会出现多对一的情况。DNS 按照一定次序将 IP 地址列表返回给客户端，浏览器使用第一个 IP 地址进行访问。开源的 Bind DNS 服务器可以满足以上要求，并能将多 IP 地址的次序改变来实现轻量级的负载均衡。高级的 DNS 功能会依据客户端地理位置返回最近的服务节点，并探测数据中心服务接口的健康情况。

企业应用有着非常高的安全要求，各层之间都会部署防火墙，直连外部的区域甚至会采用双向 NAT 防火墙。在各个节点上，所有配置文件中的密码、密钥都被要求存入集中的管控机上，接入管控机必须提前注册、授权。可以说对安全的高要求让节点的自动复制扩容变得非常困难。

Web 服务器与应用服务器也存在负载均衡关系，很多商用应用服务器通过组播通信的方式实现集群功能。在集群中有一个独立的 admin 节点，提供友好的用户界面对集群配置进行控制。这种 admin 节点是集群的“中心化”。

### 6.4.3 企业应用调整

在传统领域的应用管理中，我们的节点安全、稳定，却无法全面地自动化，实现快速扩容，同时 PaaS 上的节点异常退出的频率要高于原有环境，这是在设计应用架构时就要考虑好的高可用问题。当企业应用从传统三层模型中向 PaaS 迁移时我们必须制订一些规则：为什么要进行迁移，如何迁移。

#### 1) 为什么要进行迁移

迁移的最重要原因是应对面向互联网、面向零售终端的高并发请求访问场景。传统三层模型也能够满足要求，但其所使用的基础资源无法共享，将造成成本巨增，同时受到防火墙、“中心化”管理、安全认证等各种要求的限制，很难让扩容的动作自动化。简而言之，在先行部署架构上无法满足未来业务快速发展的场景。

从 12-Factor 中可以看出，PaaS 平台中应用的部署架构与传统三层模型存在很大的差异。在 PaaS 中没有串联的独立防火墙硬件设备，而是通过旁路的 IDS 入侵检测系统进行事后检查，并分散到节点内部软防火墙实现事前安全控制。PaaS 中的节点需要快速扩容、迁移，它不希望在新建、启动时与外部有其他联系，传统的安全注册控制限制应用的弹性能力。诸如此类的差异点还有很多，因此传统三层模型中的应用在进行迁移前还有许多内容需要调整。

#### 2) 如何迁移

传统企业应用侧重于安全性，要稳；互联网应用侧重于变化，要快。二者的需求场景起源不同，企业应用最初服务于内，而互联网从一开始就是面向终端用户的。这并不是谁优谁劣的问题，在将传统应用搬迁到 PaaS 上时，应当用两个标准来看待业务系统。

传统金融公司的 IT 系统为企业内部运转提供服务，它并不是作为渠道直接提供给最终用户使用的，其真正的渠道是线下专门的销售团队、服务团队。这种场景下 IT 系统的并发量、数据量都不会太大，功能集中在核心交易领域，关注面集中在中间件多样化，业务逻辑规则复杂性以及核心交易数据安全保障。这时可用安全稳固的标准看到企业应用是正确的。当 IT 系统发展成直接获得顾客的一部分渠道时，它就从原来的企业内部管理转变成了渠道。外部用户的使用行为和之前完全不同，其所使用的大部分功能并非直接交易，而是对公司主页、产品的浏览与查询，参与论坛评论、社交活动，这些功能中的一部分是静态的，对数据的安全要求不像交易数据那么高。但这些功能由于使用用户多，并发量大，同时为了维持用户对产品新鲜感，这些功能的变化应当是非常快的。企业应用走的是一条从

内向外的路，互联网则正好相反。如图 6-3 所示是一个电子商务交易网站提供的所有功能，大部分用户功能为浏览主页、查询产品，核心交易并不多。

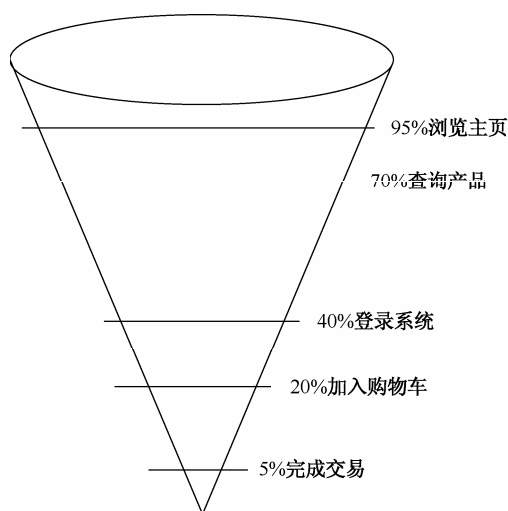


图 6-3 用户访问内容比例

迁移企业应用的第一步是外部的分而治之，将核心交易功能从系统中抽出，保持原有安全标准。而对于面向互联网的多变化、非核心模块，则应尽快放开标准，让其适应 PaaS 平台，实现快速扩容与伸缩，做好随时满足海量并发请求的准备。核心与非核心功能的划分可以在粗粒度的系统层面，也可以在细粒度的功能层面。对于作为渠道直接面向用户的新功能部分，全部采用新标准进行部署，只有在产生交易时才与内部接口进行交互。

采用分而治之的策略，筛选出可以放入 PaaS 的大多数系统与功能，我们可以将之前“厚重”的历史包袱给放下，在应用部署架构、安全准则层面对其松绑。随后，我们仍旧要思考如何让其应用内部适应 PaaS。在 PaaS 上我们不用再考虑如何在不同的服务器上配置 100 个 WebLogic Server，不用再参与到负载均衡配置中，不用再通过心跳检测来做主备切换，在原来的传统部署方式下的很多运维工作通过自动化方式解决了，但对于服务型的 PaaS 应用，我们应当有一个明确认识，它的某一个节点可能随时出现异常；它是易失性的，并不持久化任何数据，这就促使我们在构建应用时采用一种完全不同的方法。我们可以通过回答下面的问题来寻找调整应用的方法。

- 应用实例不能再依赖于本地存储持久化任何数据，日志将被发送到远程的集中平台，对其他类似情况应当如何处理？
- 如果应用接收上传文件，那么这些文件应当采用何种方式存储？
- 集群中的多实例配置文件是否完全与底层 OS 解耦，配置文件是否不依赖于主机名、IP 地址？

- 应用如何进行持久化？是保存在本地磁盘中还是保存在当前内存中？如果我们运行成百上千的实例，那么这意味着什么？
- 在应用中是否存在需要花很长时间执行的请求类型？
- 应用服务器是否采用了组播通信方式构建集群，应用服务器是否存在“中央”管理方式？

### 1) 非持久化

应用实例可以在任何计算节点运行，这就意味着它在漂移到另一个节点时，在原来的计算环境中不遗留任何影响它继续运行的持久化信息。哪些数据是持久化在本地节点的呢？日志通过远程集中来消除在本地的记录。

对于需要上传文件的应用，跨实例的数据共享传统是通过 **nas** 共享存储解决的，虽然 **nas** 可以解决多实例中共享目录的问题，但是对于应用节点经常迁移的 **PaaS** 场景，这种需要提前对操作系统进行卷挂载的方式不再适用。**PaaS** 上的常规解决方法是采用 **blob** 存储，也就是对象存储，其一般是用 **Key/Value** 方式存储文件的。这种存储提供基于 **REST API** 访问的接口，在上传成功后会返回一个唯一的 **URL** 地址用于下次将文件取回。

对象存储的第一个优势是不再需要在本地操作系统上挂载卷就能让分布式节点上的应用共享文件。其次，在使用互联网对象存储时，可能已经享受到了 **CDN**（**Content Delivery Network**）服务。企业内部也可以构建自己的对象存储服务。

### 2) 同构化

**PaaS** 上集群中的每一个实例是完全同构的，而在企业应用集群的实例中却存在或多或少的差异。应用要与底层资源解耦，配置文件中的本地主机名、主机 **IP** 地址等全部要取消。这些差异性的配置应全部从文件中移除，通过环境变量传递。另外对于依赖于 **IP** 地址的配置，应通过域名替换，才能保持实例的绝对“纯净”，将差异通过外部手段注入。

很多商用应用服务器具备管理控制台，**WebLogic**、**WebSphere** 等用于在控制台中管理集群中的所有实例。这种集中控制又加入了一层依赖关系，所有实例依赖于 **admin** 控制台，而集群中的实例需要被 **admin** 控制台区分，必然存在差异。因此对于这种“中心化”的商用应用服务器要做去“中心化”改造。可以解除应用服务器集群，直接在 **admin** 服务器上部署应用逻辑以实现扁平化，也可以更换为开源的应用服务器，例如 **Tomcat**。

### 3) 内存会话

**HTTP** 是没有状态的，用户与服务器之间的交互需要有状态，会话（**session**）的作用就在于此。应用服务器对已受信的用户在内存中创建一个会话，将用户的相关信息保存到此次会话中，并在返回的 **Http Response** 中设置一个与此会话匹配的 **Cookies ID**，在用户下次访问时通过此 **Cookies ID** 匹配所找到的会话，从而保持用户状态。需要注意的是集群前端



的负载均衡设备需要有 Cookies 识别功能，才能将相同 Cookies ID 的请求发送到同一服务器。

对于 PaaS 上的应用而言，实例异常频率增多，实例迁移时所有持久化内容也将全部消失，将会话再放在实例内存中不再可取。Tomcat 及商用应用服务器都具备会话在多实例间同步的功能模块，所有会话信息都保存在一个实例中会导致大量的内存占用。

在 PaaS 平台上常使用的会话处理方式如下。

加密 Cookies 作为会话使用。这种方式将会话数据加密后放到 Cookies 中返回给客户端，在服务端不保存任何信息。其优势是非常快，不依赖于外部服务保存数据；其缺点是在 Cookies 中并不能保存大量的数据。

我们可以通过远程的 NoSQL 如 Redis、Memcached 来存储会话信息。它同样可以享受到速度方面的优势，但使实例依赖于外部服务，在服务发现上要使用域名加环境变量的方式解决。

#### 4) 异步调用

请求并发量高并不是服务器堵塞的唯一原因，就某些处理时间过长的任务，即便应用服务器有足够空闲的线程，就单个处理请求来说任务依然处于堵塞状态，这时请求量稍微增加即可让一个应用服务器完全瘫痪。在 PaaS 平台上要避免将这种长任务操作直接暴露给用户。最常见的方法是通过消息队列将长任务与用户请求解耦。每次请求到来之际并不进行实时处理，而是将任务放入后端消息队列中，之后立即返回给用户。后台工作进程从队列中取出任务进行执行。待处理完毕后通过邮件或者其他方式通知用户查看。在 PaaS 中有必要放置一个公共的即取即用的分布式队列。

# 计算单元 Docker

## 7.1 Docker介绍

Docker 在 PaaS 平台中很好地充当了计算单元打包的角色，“Build, Ship and Run Any App, Anywhere”是 Docker 官网的标语，表示 Docker 可以“一次构建，任意运行”。Docker 为什么这么神奇，它由什么构成，本章将对其进行详细的讲解。

### 7.1.1 Docker是什么

无法找到一个最贴切的事物来形容 Docker，你可以把它简单地理解为一个虚拟机，它是一个在近几年来非常火的轻量级容器技术，它的轻量级体现在专注于 Linux 平台，使用 Linux 中诸如 namespace、cgroups 等内核特性，让进程运行在一个隔离的环境中，如同虚拟机一样拥有独立的操作系统空间。

Docker 天生为构建 PaaS 平台而生，它诞生于 DotCloud 公司，这家公司的研究方向就是云及 PaaS。在同一时期，几家大的云服务提供商都基于 Linux 容器技术做好了各自的封装，这对 DotCloud 来说市场竞争压力不小，在发现越来越多的开发者关注于容器技术的完整封装方案时，DotCloud 以开源的方式将 Docker 对外发布，并声明会完全专注于 Docker 技术的研发，持续支持与改进 Docker，很快，DotCloud 随着 Docker 的开源而闻名。

在互联网上可以找到很多关于构建分布式应用的工具、技术，但从来没有一样像 Docker 这样快速流行起来，这主要在于 Docker 的设计在保证基本需求的前提下做到了足够简捷，这个基本需求保证了 Docker 可以在各类 Linux 操作系统上运行（在 Windows、OSX 上通过放置虚拟机也可以运行），并保证有绝对的兼容性。应用程序所依赖的环境可以在桌面、虚拟机、数据中心服务器、云上无差异地运行而不用担心配置差异，这对开发人员来说是一大福音。同时 Docker 天生的操作友好性让运维人员可以很快上手，“一次构建，任意运行”的工作流程让开发人员关注于应用程序的构建，而运维人员则关注于将构建好的镜像部署到当前的环境中，大大简化了代码管理与版本发布在开发、运维人员之间的交互工作。

刚刚说的这些功能不是用虚拟机模板也能够实现吗？Docker 和虚拟机到底有什么区别呢？虚拟机技术相对于容器来说要“重”很多，虚拟机采用 Hypervisor 技术，是虚拟机与物理服务中间的一层，为每一台虚拟机分配适量的内存、CPU、网络和磁盘，并加载所有虚拟机的客户操作系统，这种虚拟化意味着很大部分的计算、存储资源被使用在冗余的客户操作系统上，启动过程要远慢于容器。

Docker 采用了一种截然不同的方法，它直接采用 Linux 的容器技术来隔离进程，让其认为自己运行在一个单独的操作系统中，而实际上仍然运行在同一个操作系统中，共享同一个内核，资源利用率远高于 Hypervisor。有得必有失，Docker 基于容器的虚拟化注定它只能运行在 Linux 操作系统上，而且目前只支持在 64 位操作系统上运行。Docker 在文件系统上使用了分层结构的 AUFS（Another Unionfs）文件系统，将存储资源的共享也实现了最大化，进一步保证了资源使用率的提升。最大化的资源共享意味着进程间的隔离性下降，在这方面，Linux 广泛使用的进程资源管理方案如 namespaces、cgroups 被集成到了 Docker 之中，从而实现了与虚拟机类似的隔离性。

如图 7-1 所示，我们看到容器可以与主机上的其他容器、进程共享基础资源，而在虚拟机的每个实例中却要运行一个完整的操作系统。

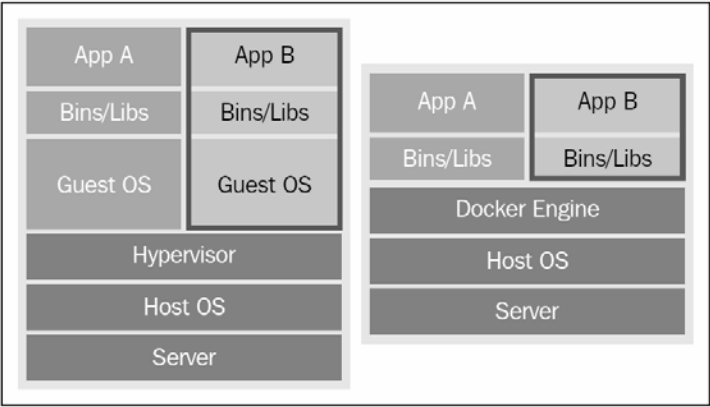


图 7-1 Docker 技术和 Hyperviisor 虚拟机技术的对比

7.1.2 Docker术语

Docker 有三个重要术语能够帮助我们快速了解它的整个运转工作流，它们分别是镜像（image）、容器（container）和仓库（repositories）。

1. 镜像

镜像类似于虚拟机的模板，可以将其看作静态文件。我们可以想象 Linux 内核是 0 层，当我们构建一个 Docker 镜像时，其覆盖到了内核层上，这个镜像 是 1 层，也可称之为 base

image，镜像是只读的，不能够被修改。Docker 镜像能够构建在另外一个镜像之上，如同乐高积木一样一层一层地叠加。前一个镜像被称为后一个镜像的 `parent image`，它们继承了父镜像的所有属性与设置。

Docker 镜像有一个镜像标识符，它是一个 64 字符长度的十六进制字符串，但是我们在使用镜像的过程中几乎从来不会用到这个 ID，而是通过镜像名称访问的，其是一个供人阅读的名字和标签对。镜像可以命名为 `ubuntu:latest`、`ubuntu:precise`、`django:1.6`、`django:1.7` 等。对镜像的修改有两种方法：在一个文件中指定一个基础镜像及需要完成的修改；或通过“运行”一个镜像，对其进行修改并提交。

Docker 生成镜像的方式有两种：一种是基于活动的容器来生成，另一种则是在原有镜像上通过文件的描述说明来生成。Dockerfile 就是 Docker 中专门定义的一种领域特定语言（DSL，Domain Specific Language），其包含了用于生成镜像的指令集，我们可以把它看作 Docker 中的 Makefile 文件。

Docker daemon 是操作系统上的 Docker 管理进程，它负责对机器上的镜像、容器进行管理，该进程需要 root 管理员的权限才能运行，对外提供一个 RESTful 的 API 接口。

我们很多时候对 Docker client 与 Docker daemon 区分不清，实际上真正进行管理的是 Docker daemon 进程，而 Docker client 是提供给系统管理员访问 Docker 的操作接口，它与 Docker daemon 之间也是通过 RESTful API 进行交互的。

## 2. 容器

一个 Docker 容器在我们执行 `Docker run <image>` 命令时创建，它在镜像上加入了可写的一层，在这一层上可以启动一个进程，并且具有两个不同的状态：运行（Runing）或退出（Exited）。当我们用 Docker run 命令启动一个容器时，它会一直保持为运行状态，直到自己退出或者被我们停止，从而进入退出状态。在操作系统中所有活动的计算逻辑都是由进程表示的，在 Docker 中要启动一个进程必须从容器开始。在容器启动过程中的所有修改会全部写回到文件系统中，在容器停止后这部分内容不会丢失，所有内容都将写入容器的文件系统层中，而不是它的下层的镜像。

## 3. 仓库

Docker 能够快速流行的原因在于其有一个公网中央仓库，你可以很方便地在这里找到你所需要的镜像，Docker 官网的中央仓库地址是 <https://hub.docker.com/>。

仓库如同你的版本库，你可以将计算逻辑的各个版本放在此处，之后在分散的计算节点下载并运行。由于其镜像独立的“乐高”积木式层叠方式，每一个小版本的修改所占用的存储空间并不大。除了使用公网的中央仓库，我们也可以构建自己的私有仓库。

镜像、容器、仓库三者浑然地结合在一起，形成了 Docker 独有的工作流程，从仓库中申请一个 base image，创建一个容器运行起来，之后安装软件，配置参数，再将其打包成一

个新的镜像，随后推送到仓库中，将一个新的计算逻辑共享给所需之处使用。

### 7.1.3 Docker安装

#### 1. Ubuntu Trusty 14.04 LTS

对 Docker 支持度最好的 OS 版本就是 Ubuntu Trusty 14.04 LTS 了，我们可以使用包构建工具 apt-get 快速地完成安装任务。

我们打开一个终端，逐行执行下面的命令：

```
sudo apt-get update
sudo apt-get install docker.io
source /etc/bash_completion.d/docker.io
```

在这里我们首先更新 apt-get 包管理器的列表清单，获取所有的包、版本，以及依赖关系信息。随后开始安装 Docker，最后使用 Source 命令在当前 Shell 环境下读取与执行 Docker.io 文件中的命令，它会设置相关环境变量。

#### 2. Red Hat Enterprise Linux 7

红帽企业版 Linux7 支持 Docker，Docker 的安装源放在了红帽的附加仓库中，在这个 yum 源中的安装包有着不同的服务条款，读者可以自行查找浏览。

在安装好 RHEL7 之后，使用红帽的订阅管理工具来安装 Docker，首先运行从附加仓库、可选仓库中下载的软件包：

```
# subscription-manager repos --enable=rhel-7-server-extras-rpms
# subscription-manager repos --enable=rhel-7-server-optional-rpms
```

在设置好订阅管理器的相关参数后，可以直接使用 yum 来安装 Docker，在这里我们将 Docker 及 Docker 仓库安装在同一台服务器上，实际上 Docker 仓库是用来保存、维护所有镜像的地方，通过这种安装方式，我们可以快速地建立自己的私有仓库，而 Docker 需要在所有提供 Docker 服务的机器上安装：

```
# yum install docker docker-registry
# yum install device-mapper-libs device-mapper-event-libs
```

因为 RHEL7 的防火墙服务于 Docker 服务，有冲突存在，因此在安装之后我们需要将防火墙服务关闭：

```
# systemctl stop firewalld.service
# systemctl disable firewalld.service
```

最后我们启动 Docker 服务，运行 Docker 服务，并查看 Docker 的服务状态：

```
# systemctl start docker.service
# systemctl enable docker.service
# systemctl status docker.service
```

```
Docker.service - Docker Application Container Engine
  Loaded: loaded (/usr/lib/systemd/system/Docker.service; enabled)
  Active: active (running) since Thu 2014-10-23 11:32:11 EDT; 14s ago
    Docs: http://docs.Docker.io
  Main PID: 2068 (docker)
    CGroup: /system.slice/Docker.service
            └─2068 /usr/bin/docker -d --selinux-enabled -H fd://
  ...
```

## 7.2 Docker容器命令

在本节我们首先通过命令行做一些简单的 Docker 操作，之后详细地介绍 Docker 命令行及远程 API 接口。

1) 从公网仓库中获取一个 Ubuntu 最新的 base 镜像。

```
$ docker pull ubuntu:latest
```

2) 从 Ubuntu 镜像运行一个容器，执行 bash 命令。

```
$ docker run -dt ubuntu:latest bash
```

3) 查看当前主机下的所有镜像。

```
$ docker images
```

4) 查看当前主机下所有容器的状态。

```
$ docker ps -a
```

以上是 Docker 命令行最频繁使用的几条命令，通过以上命令我们可以运行并观察一个 Ubuntu 容器。

### 7.2.1 run命令

Docker 容器有一个清晰的生命周期状态，我们常常直接使用 run 命令从一个镜像生成运行状态（Running）的容器，而实际上容器通过 create、start、stop、rm 命令来管理自己的生命周期。create 创建一个容器，这时容器并没有任何状态，start 启动一个新建的或者停止的容器，stop 停止一个运行的容器、rm 将一个停止的容器从主机上删除。

run 命令使用的基本方法如下：

```
$ docker run [options] IMAGE [command] [args]
```

run 命令所涉及的选项及其说明如表 7-1 所示。

表 7-1 run 命令所涉及的选项及其说明

选 项	说 明
-a, --attach=[]	将当前的标准输入 stdin、输出 stdout 或错误输出 stderr 指定到容器中，在将容器放到后台运行时这个选项无效
-d, --detach	将容器放到后台运行
-i, --interactive	采用交互模式运行容器，保持 stdin 标准输入文件为打开状态
-t, --tty	分配一个伪终端标识符，这在你登录容器时需要打开
-p, --publish=[]	容器内的端口服务在主机 OS 上是无法访问的，这就需要提前对外发布端口，我们也可以认为是端口映射（ip:hostport:containerport）
--rm	在容器运行完毕退出后自动删除容器（这个选项不能与-d 同时使用）
-v, --volume=[]	容器中的数据会随着容器生命周期的结束而消失，我们可以通过该选项将外部存储映射到容器内，将外部数据给容器访问，或者将容器的数据保存到外部（/host:/ container）
--volumes-from=[]	从另外一个容器中 mount 卷
-w, --workdir= " "	设置容器中的工作文件夹
--name= " "	分配一个容器名字，如果不指定，则会自动生成
-h, --hostname= " "	分配一个主机名
-u, --user= " "	指定容器运行后的 uid 或用户名
-e, --env=[]	容器内的环境变量，在容器差异化时可以通过这个选项为相同的镜像容器设置不同的环境变量
--env-file=[]	从一个行级的文件中读取环境变量
--dns=[]	设置容器的 DNS 服务器
--dns-search=[]	设置容器的 DNS 查询域
--link=[]	在同一主机上容器可以通过 Link 链接进行访问，从而无须暴露端口（name:alias）。
--cpuset= " "	通过 cpuset 让容器运行在指定的 CPU 上
-c, --cpu-shares=0	cpu-shares 是一个权重值，当多个容器运行在相同的 CPU 资源上时，会依据此权重值进行资源分配
-m, --memory= " "	指定容器所分配的内存大小

在使用 run 命令行运行容器时，-t、-i 这两个选项能够让我们通过终端进入容器中进行交互式命令行操作，如果不带这两个选项，那么我们将无法通过 attach 命令进入容器中，容器内的进程处于不可观测的状态。-d 选项让容器作为后台进程运行，我们在容器中运行服务器时会常常用到。

每次创建一个容器，容器会被分配到一个 64 位 char 字符的唯一 ID，另外还会分配一个容器名，这个名称可通过-name 选项自定义，也可随机生成。值得我们注意的是每次以 ID 操作一个容器时，并不需要我们输入完整的 ID 名称，通过 ID 字符串中的最开始的两个字

## PaaS 实现与运维管理：基于 Mesos + Docker + ELK 的实战指南

符就可以定位到容器。

```
$ docker run -dit --name my-first-ct ubuntu /bin/bash
1a41fa96c38836df8a809049fb3a040db571cc0cef000a54ebce978c1b5567ea
$ docker attach my-first-ct
root@1a41fa96c388:/#
```

在容器中的进程运行完成后，容器就退出了，这时容器并没有消失，而是作为 **exited** 状态存在着，我们可以通过 **start** 命令再次启动容器。**ps** 命令能查看主机上所有容器的信息。你可以在启动容器时加入 **--rm** 选项，这样进程退出时容器会直接被删除。

```
$ docker ps -a
CONTAINER ID IMAGE          COMMAND          CREATED    STATUS    NAMES
1a41fa96c388 ubuntu:latest bash          1 hour ago Exited    my-first-ct
```

接下来我们运行一个 **Python** 容器，启动后执行 **Python** 脚本运行一个 **HTTP** 服务器。

```
$ docker run -it --name my-Pythonserver-1 --rm Python:2.7 \
Python -m SimpleHTTPServer 8000;
Serving HTTP on 0.0.0.0 port 8000
```

每一个容器都动态分配到了一个 **172.17.0.0/16** 的私网 IP，启动后的 **HTTP** 服务器无法提供给外部访问，这里的外部指的是同一个 **OS** 内外的进程。这时 **-p** 选项可以发挥作用，它将容器内的端口绑定到 **OS** 的端口上，这样外部也就能够访问容器内的服务了。

```
$ docker run -p 0.0.0.0:8000:8000 -it --rm --name my-Pythonserver-1 \
Python:2.7 Python -m SimpleHTTPServer 8000;
Serving HTTP on 0.0.0.0 port 8000 ...
172.17.42.1 - - [24/Jul/2014 18:25:46] "GET / HTTP/1.1" 200 -
```

在这个例子中我们将端口通过 **8000** 映射出来，外部访问 **OS** 的 **8000** 端口即容器中的服务。

为了提高容器的安全性，避免潜在的风险，**Docker** 默认在容器中关闭了内核的一些功能，一般场合并不要求使用这些功能，我们可以通过 **-privileged** 选项打开。

在容器中读写的数据不会在 **OS** 中体现，这些数据会随着容器生命周期的消亡而消亡，有一些需求会需要将这些数据保存下来，或者将 **OS** 上的数据提供给容器使用，**--volumes** 选项能够将一个 **OS** 上的外部卷 **mount** 到容器中。

```
$ docker run -v /wls/Python:/wls/Python:wr -w /wls/Python -p
0.0.0.0:8000:8000 -it \ --name OD-Pythonserver-4 Python:2.7 Python -m
SimpleHTTPServer 8000;
Serving HTTP on 0.0.0.0 port 8000 ...
172.17.42.1 - - [24/Jul/2014 18:33:45] "GET / HTTP/1.1" 200 -
```

在上面的 **Docker** 命令中，我们使用 **-v** 将外部 **OS** 的卷 **mount** 发送给容器使用，在 **-v** 选项的最后面我们可以加入 **wr** 或者 **ro** 来设定卷的读写权限。容器中默认的工作目录是根目录，在这里设置 **-w** 选项，指定工作目录为 **/wls/Python**。

**Docker 1.2** 之后的版本，除了能够 **mount** 一般的文件系统，还支持对设备映射绑定，这



需要使用--device 选项。

最后对于容器中运行 **Service** 型的应用而言，容器内的进程都是长时间运行的，在 Docker1.2 之后的版本中加入了--restart 选项来设定容器的重启策略。

- **no**: 在容器消亡退出后不重启（默认）。
- **on-failure**: 在容器退出后，如果退出码不是 0，则重启容器。它也支持一个退出码的最大值设置，例如 on-failure:5。
- **always**: 无论容器的退出码是什么，都将重启。

## 7.2.2 start命令

在 **Docker run** 命令中我们看到容器从一个镜像中创建并启动，在容器退出后且处于 **exited** 状态时，我们可以通过 **start** 命令再次将容器运行起来。

我们在前面讨论过容器与虚拟机的区别，通过上面的操作我们会发现容器并不是一个完整的操作系统，而是在启动容器时由 **Shell** 命令执行的程序所启动的一组进程，当这组进程全部退出后这个容器也就跟着退出了。在 **Docker** 官方网站上经常用 **bash** 这个命令作为例子，因为这是一个交互型的进程，在设置-i、-t 选项后启动容器，我们会立马进入 **bash Shell** 的交互中来，在感官上会认为整个 **OS** 已经启动，在 **Shell** 中输入 **exit** 时，这个 **bash** 进程也跟着退出了，容器的状态变为 **exited**。

很多时候我们在容器中执行的 **Shell** 并不是 **bash**，而是我们的应用服务器程序，例如 **WebLogic**、**Tomcat** 等，为了方便管理，普遍的情况是在 **Shell** 执行脚本中再启动一个 **sshd** 进程，用以模拟远程登录到容器中检查状况。

**Docker start** 的选项基本上与 **run** 命令一样。

```
$ docker start [-i] [-a] <container(s)>
```

下面通过 **ps** 命令列出所有 **Docker** 容器，从中启动一个 **exited** 状态。

```
$ docker ps -a
CONTAINER ID IMAGE          COMMAND          CREATED   STATUS    NAMES
e3c4123cff  ubuntu:latest Python -m 1h ago   Exited    my-Pythonserver-2
81bb2a92ab0c ubuntu:latest /bin/bash 1h ago   Exited    kindly_river
d52fef570d6e ubuntu:latest /bin/bash 1h ago   Exited    fly_skywork
eb424f5a9d3f ubuntu:latest /bin/bash 20h ago   Exited    my-name-example
$ docker start -ai my-Pythonserver-2
Serving HTTP on 0.0.0.0 port 8000
```

### 7.2.3 stop命令

容器执行一个 Shell 命令后会由此带动一组进程，进程结束后容器自动退出，在某些情况下这些进程是服务类型的，会一直在监听，等待访问请求的到来。这类进程是长任务行进程，如果要停止掉容器，那么可以使用 stop 命令，它会向容器发送一个 SIGTERM 信号，容器内的进程收到这个信号后会调用自己的 shutdown 程序，这种停止方式很优雅，在一定的时间后如果容器还没有停止，则 Docker 会继续发送一个 SIGKILL 信号，强制性地停止容器。

```
$ docker run -dit --name my-stop-example ubuntu /bin/bash
$ docker stop my-stop-example
```

stop 命令强迫停止之前的等待时间周期可以通过 -t 选项进行设置。

### 7.2.4 restart命令

restart 命令将一个运行中的容器进行重启，大部分命令选项与 run 相同。

### 7.2.5 attach命令

attach 命令常常让人们进入一个误区，类似于登录一个虚拟机 OS。如前所述，要使用 attach 附属到一个容器上，在启动容器前要将 -i、-t 加上，在启动时也要将 -d 选项加上，daemon 容器在后台运行。attach 附属到一个容器上，它如同将这个后台任务切换到前台，将用户终端的标准输入 stdin、标准输出 stdout，以及错误输出 stderr 指向容器进程中的这三个文件描述符中。

```
$ docker run -dit ubuntu:latest bash
e5b89df449612fd2b2eb0518c0b890e5ef1097d80fbb710054374f3be918b130
$ Docker attach e5b8
[root@e5b89df44961 /]#
```

### 7.2.6 ps命令

ps 命令用来列出所有的容器信息，它使用的方式如下：

```
$ docker ps [option(s)]
```

ps 命令所涉及的选项及说明如表 7-2 所示。

表 7-2 ps 命令所涉及的选项及说明

选 项	说 明
-a, --all	显示所有的容器，包括已经停止的容器
-q, --quiet	仅显示容器 ID

续表

选 项	说 明
-s, --size	打印容器的空间大小
-l, --latest	仅显示最新启动的容器
-n= " "	显示最近运行的 n 个容器，包括停止了容器
--before= " "	显示在某个容器 ID 之前启动的所有容器，包括停止的容器
--after= " "	显示在某个容器 ID 之后启动的所有容器，包括停止的容器

7.2.7 inspect命令

inspect 命令允许你获取一个容器、镜像的详细信息，它会返回一个 JSON 格式的数据。

```
$ docker inspect rhel65
```

```
[{
  "Architecture": "amd64",
  "Author": "",
  "Comment": "",
  "Config": {
    "AttachStderr": false,
    "AttachStdin": false,
    "AttachStdout": false,
    "Cmd": [
      "/bin/sh",
      "-c",
      "sh /root/start.sh"
    ],
    "CpuShares": 0,
    "Cpuset": "",
    "Domainname": "",
    "Entrypoint": null,
    "Env": [
      "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/",
      "LANG=en_US.UTF-8",
      "LC_ALL=en_US.UTF-8"
    ],
    "ExposedPorts": {
      "22/tcp": {}
    },
    "Hostname": "5d202c35fc0d",
    "Image": "687d3ce3560f0abdea581b9fe9ca4657b1edc6bab22466a6738def295cbc5bdf",
    "MacAddress": "",
    "Memory": 0,
    "MemorySwap": 0,
```

```

        "NetworkDisabled" : false,
        "OnBuild" : [],
        "OpenStdin" : false,
        "PortSpecs" : null,
        "StdinOnce" : false,
        "Tty" : false,
        "User" : " ",
        "Volumes" : {},
        "WorkingDir" : " "
    },
    "Container" : " 5d202c35fc0df8028f3dc1e8f24f5fa03bf878e4577856197bc
82e9d092200bd ",
    "ContainerConfig" : {
        "AttachStderr" : false,
        "AttachStdin" : false,
        "AttachStdout" : false,
        "Cmd" : [
            "/bin/sh",
            "-c",
            "#(nop) CMD [/bin/sh -c sh /root/start.sh]"
        ],
        "CpuShares" : 0,
        "Cpuset" : " ",
        "Domainname" : " ",
        "Entrypoint" : null,
        "Env" : [
            "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/
sbin:/bin",
            "LANG=en_US.UTF-8",
            "LC_ALL=en_US.UTF-8"
        ],
        "ExposedPorts" : {
            "22/tcp" : {}
        },
        "Hostname" : " 5d202c35fc0d ",
        "Image" : " 687d3ce3560f0abdea581b9fe9ca4657b1edc6bab22466a
6738def295cbc5bdf ",
        "MacAddress" : " ",
        "Memory" : 0,
        "MemorySwap" : 0,
        "NetworkDisabled" : false,
        "OnBuild" : [],
        "OpenStdin" : false,
        "PortSpecs" : null,
        "StdinOnce" : false,
        "Tty" : false,
        "User" : " ",
        "Volumes" : {},
        "WorkingDir" : " "
    }
}

```

```

    },
    "Created": "2015-04-13T09:38:32.753201072Z",
    "DockerVersion": "1.5.0",
    "Id": "78ecdbfa7073afefeb5487777a14f8b8177d0ccf27b45eef8639f5503249e63d",
    "Os": "linux",
    "Parent": "687d3ce3560f0abdea581b9fe9ca4657b1edc6bab22466a6738def295cbc5bdf",
    "Size": 0,
    "VirtualSize": 364136377
  }
]

```

## 7.3 Docker镜像命令

7.2 节的内容让我们能够在既定的镜像上运行程序，而在实际工作中我们需要在原有镜像上做客户化的修改，镜像定制化后运行。一般的工作流是从仓库中搜索镜像，选择自己的镜像并下载，编辑镜像进行客户化，之后依据这个镜像产生容器。另外，为了实现在其他地方运行容器，将生成好的新镜像上传到仓库也非常重要。

客户化镜像的方式一般有两种：一种是基于容器生成镜像，另一种是用 Dockerfile 文件通过 `build` 命令构建镜像。本节将着重介绍第一种方法，在 7.4 节中我们将讲述 Dockerfile。

如图 7-2 所示的是 Docker 使用的整个工作流。

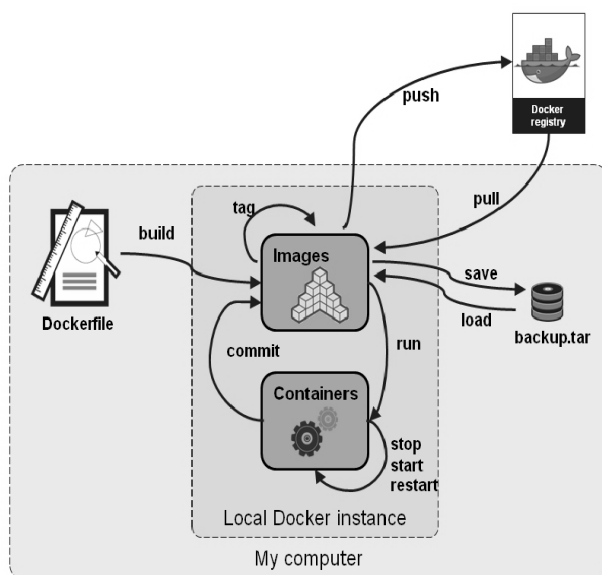


图 7-2 Docker 工作流展示

### 7.3.1 search、pull、push命令

`search` 命令用来在仓库中搜索我们需要的镜像，下面的代码用于从仓库中搜索 MySQL 相关的镜像：

```
$ docker search mysql | less
```

`pull` 命令用来从仓库中抓取镜像到本地文件上，默认从公有仓库中抓取，但是你也可以指定到私有仓库里：

从公有仓库抓取 MySQL 镜像的代码如下：

```
$ docker pull mysql
```

从公有仓库抓取带标签的 MySQL 镜像的代码如下：

```
$ docker pull mysql:latest
```

如果想从私有仓库抓取镜像，那么可以在镜像名称前加上仓库地址：

```
$ docker pull <path_to_registry>/<image>
```

`push` 命令与 `pull` 相对应，它将本地的一个镜像推送到仓库中：

```
$ docker push NAME[:TAG]
```

### 7.3.2 commit命令

我们可以通过 `commit` 命令来客户化镜像，它从我们现有的一个容器中生成一个新的镜像，你可以在镜像中写入自己的备注信息。如表 7-3 所示。

表 7-3 `commit` 命令所涉及的选项和说明

选 项	说 明
<code>-p, --pause</code>	在提交之前将容器暂停
<code>-m, --message= " "</code>	添加用来描述镜像的备注信息
<code>-a, --author= " "</code>	添加作者信息

我们通过一个例子来学习从容器中生成镜像的方法。首先运行一个 Ubuntu 发行版的 Linux，命令行使用 `bash`：

```
$ docker run -name my.ubuntu -it ubuntu /bin/bash
[ root@3b0d5a04cdcd:/]$ apt-get -y install mysql
```

另一个终端在这个容器之上生成镜像：

```
$ docker commit -m " my.mysql - install mysql database on ubuntu " -
a "yuhe yuhehome@sina.com" my.ubuntu yuhe/code.it:v1
```

7.3.3 image、diff、rmi命令

image 命令用来显示当前主机下的所有镜像信息，如表 7-4 所示。

表 7-4 image 命令所涉及的选项和说明

选 项	说 明
-a, --all	显示所有镜像，包括中间层
-f, --filter=[]	提供了一些镜像过滤值
--no-trunc	不剪裁任何信息，显示完整的镜像 ID
-q, --quiet	仅显示镜像 ID

diff 命令用来比较当前容器与其启动镜像之间的文件差别：

```
$ docker diff boring_hopper
C /root
C /root/.bash_history
```

rmi 命令用来删除镜像，在删除一个镜像时会将其下层的所有镜像全部删除：

```
$ docker rmi pingan_ep
```

7.3.4 save、load、export、import命令

save、load、export、import 四个命令总的来说是将镜像保存为.tar 格式的文件，以及向.tar 格式的文件导入生成镜像。

save 命令将镜像输出到 stdout 标准输出中，以.tar 压缩格式保存文件，在文件中保留了镜像的所有层级及元数据信息：

```
$ docker save -o pingan_ep.tar pingan_ep.it
```

-o 选项将输出从标准输出重定向到一个文件中，这种方式经常用来对镜像备份，之后使用 load 命令来加载这份压缩文件。

load 命令从.tar 压缩文件中加载镜像，这种方式会将镜像的层级及元数据信息全部还原：

```
$ docker load -i pingan_ep.tar
```

-i 选项指定具体的文件名，上例是从我们压缩的.tar 文件中还原镜像。

export 命令与 save 命令的不同之处在于它直接从一个容器中导出压缩文件，同时，这个文件是以扁平方式存在的，在这个文件中镜像的层级、元数据信息等都将融合一个文件中，其关联的历史信息将会丢失：

```
$ sudo Docker export yuhe_redis > yuhe_redis.bak.tar
```

在这里我们将一个名为 `yuhe_redis` 的容器导出到压缩文件中。

`import` 命令创建一个空的镜像，之后从压缩文件中导入内容，它与 `export` 命令是对应的，`import` 命令可以从远程 URL 和本地文件系统中导入文件：

```
$ docker import http://example.com/test.tar.gz
$ cat yuhe_redis.bak.tgz | Docker import - yuhe_redis:imported
```

以上两项分别从远程和本地导入压缩文件，生成镜像。

## 7.4 Docker网络与链接

Docker 容器的网络、数据卷部分的内容需要从命令中拿出来单独讨论，这两部分内容决定了在 PaaS 平台中 Docker 如何对外部提供服务，并共享、存储数据。

### 7.4.1 Docker网络模式

Docker 的网络设置直接决定了我们如何向外暴露服务，在 7.2.1 节中可看到我们在 `run` 命令中可以使用 `--publish` 将容器内的端口映射到主机上，这是 Docker 网络设置默认的 bridge 网桥的方式，在启动容器时可以通过 `--net` 来设置网络模式。Docker 的网络模式有以下四种。

- host 模式：使用 `--net=host` 指定。
- container 模式：使用 `--net=container:NAME_or_ID` 指定。
- none 模式：使用 `--net=none` 指定。
- bridge 模式：使用 `--net=bridge` 指定，默认设置。

Docker Daemon 启动后会在服务器上创建一个名为 `Docker0` 的虚拟网桥，让我们通过一系列命令查看 `Docker0` 网桥的 IP 地址设置、路由设置。

可通过 `ifconfig` 查看 `Docker0` 的网卡设置，它被分配了一个 `172.17.42.1/16` 的 IP 地址，而 `172.17.0.0/16` 的整个私有网络 IP 段将保留给主机上所创建的容器使用，默认的网络模式为 bridge 的容器在启动后都会有一个该网段 IP 地址的网卡：

```
root@ubuntu:~# ifconfig Docker0
Docker0  Link encap:Ethernet  HWaddr 56:84:7a:fe:97:99
          inet addr:172.17.42.1  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::5484:7aff:fefe:9799/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:10 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
```



```
collisions:0 txqueuelen:0
RX bytes:676 (676.0 B) TX bytes:648 (648.0 B)
```

让我们启动一个 `bash` 命令的容器，并 `attach` 进入查看容器网络设置：

```
root@ubuntu:~# docker run --name yuhe_network -idt ubuntu:latest
/bin/bash
root@ubuntu:~# Docker attach yuhe_network
root@c08930800400:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 46:0b:1a:68:d7:83
          inet addr:172.17.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::440b:1aff:fe68:d783/64 Scope:Link
          UP BROADCAST RUNNING MTU:1500 Metric:1
          RX packets:16 errors:0 dropped:0 overruns:0 frame:0
          TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1296 (1.2 KB) TX bytes:816 (816.0 B)
```

```
root@c08930800400:/# netstat -rn
Kernel IP routing table
Destination        Gateway            Genmask           Flags   MSS Window
irrtt Iface
0.0.0.0            172.17.42.1       0.0.0.0           UG      0 0
0 eth0
172.17.0.0         0.0.0.0           255.255.0.0       U       0 0
0 eth0
```

可以清楚地看到容器自动配置的 `eth0` 网卡的 IP 地址为 `172.0.0.2/16`，并且将默认网关指向了外部的 `Docker0`：

```
root@c08930800400:/# ping 172.17.42.1
PING 172.17.42.1 (172.17.42.1) 56(84) bytes of data.
64 bytes from 172.17.42.1: icmp_seq=1 ttl=64 time=0.288 ms
64 bytes from 172.17.42.1: icmp_seq=2 ttl=64 time=0.082 ms
```

在容器中 `ping` 网关是相通的，容器如果要访问外部，那么其数据包也会通过网桥 `Docker0` 转发到外部。我们再在主机上用 `ifconfig` 命令查看所有的网络设备信息，你会发现容器启动后有一个虚拟的网络设备：

```
veth35e3  Link encap:Ethernet  HWaddr d6:e9:43:cb:76:bc
          inet6 addr: fe80::d4e9:43ff:fe68:d783/64 Scope:Link
          UP BROADCAST RUNNING MTU:1500 Metric:1
          RX packets:22 errors:0 dropped:0 overruns:0 frame:0
          TX packets:28 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1824 (1.8 KB) TX bytes:2304 (2.3 KB)
```

`bridge` 网桥模式如图 7-3 所示。

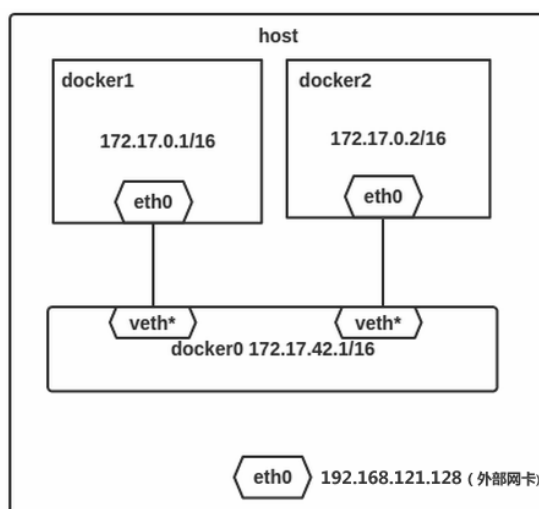


图 7-3 bridge 网桥模式的示意图

启动一个容器时，会在容器内部、主机上创建一对网络设备，在容器内部为 `eth0`，而外部则是 `veth*` 网络设备，这个设备加在了主机上的 `Docker0` 网桥之中，从而实现了容器与外部的通信。

我们再连续启动两个容器，通过 `brctl` 命令查看网桥信息：

```
root@ubuntu:~# docker run -idt ubuntu:latest /bin/bash
81cd1d943ad5809a80278f9713f384e983457313bab8a47a395f58752a4ed6ab
root@ubuntu:~# docker run -idt ubuntu:latest /bin/bash
d21fb87275each4abde0ec7341052bd4a85301ca143bfd4c4ff37ffdc7e4a2ea

root@ubuntu:~# brctl show
bridge name      bridge id          STP enabled      interfaces
Docker0          8000.56847afe9799  no               veth35e3
                  veth64b2
                  vethb7f5
```

在 Ubuntu 上安装 `brctl` 工具时使用如下命令：

```
root@ubuntu:~# apt-get install bridge-utils
```

Docker 使用了 Linux 的 Namespace 技术来进行资源隔离，PID Namespace 从进程级别隔离资源，Mount Namespace 用于隔离文件系统，Network Namespace 则用于隔离网络。一个 Network Namespace 有一份独立的网络环境。在 Docker 容器默认的 bridge 模式中容器有独立的 Network Namespace。

如果使用 host 模式，那么容器将和主机公用一个 Network Namespace，不会虚拟自己的网卡，配置自己的 IP 地址等，而是保持与主机一致。

我们用 host 模式启动一个容器，通过 `ifconfig` 查看网卡信息会发现与外部主机一模一样：

```
root@ubuntu:~# docker run -idt --net=host ubuntu:latest /bin/bash
1eb563243ad5809a80278f9713f384e983457313bab8a47a395f58752a4ed6ab
```

```
root@ubuntu:~# docker attach 1e
```

下面是我们进入容器 `bash` 中看到的网络设置：

```
root@ ubuntuContainer:/# ifconfig
Docker0    Link encap:Ethernet  HWaddr 56:84:7a:fe:97:99
            inet addr:172.17.42.1  Bcast:0.0.0.0  Mask:255.255.0.0
            inet6 addr: fe80::5484:7aff:fefe:9799/64 Scope:Link
            UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
            RX packets:39  errors:0  dropped:0  overruns:0  frame:0
            TX packets:20  errors:0  dropped:0  overruns:0  carrier:0
            collisions:0 txqueuelen:0
            RX bytes:2664 (2.6 KB)  TX bytes:1656 (1.6 KB)

eth0       Link encap:Ethernet  HWaddr 00:0c:29:00:cb:c5
            inet addr:192.168.121.128  Bcast:192.168.121.255  Mask:255.
255.255.0
            inet6 addr: fe80::20c:29ff:fe00:cbc5/64 Scope:Link
            UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
            RX packets:180483  errors:0  dropped:0  overruns:0  frame:0
            TX packets:63024  errors:0  dropped:0  overruns:0  carrier:0
            collisions:0 txqueuelen:1000
            RX bytes:213187297 (213.1 MB)  TX bytes:4659711 (4.6 MB)
```

我们在容器内安装一个 `SSHd`，之后启动服务：

```
root@ubuntuContainer:/# service ssh start
root@ubuntuContainer:/# service ssh status
```

编辑 `/etc/SSH/SSHd_config` 文件，将容器上的 `SSHd` 服务端口修改为 1234，启动 `sshd`：

```
root@ubuntu:/# service ssh restart
```

现在可以使用主机上的 `eth0` 网卡 IP 地址（笔者使用的虚拟机的地址为 192.168.121.128）通过 1234 端口登录容器了。

`contain` 模式与 `host` 模式类似，但它不与主机共享 `Network Namespace` 地址空间，而是与一个指定的容器共享。

`none` 模式拥有自己的 `Network Namespace`，但不为容器进行任何网络配置，需要用户自行添加。

## 7.4.2 pipework管理网络

由于在 `PaaS` 平台中我们会将所有容器等同于一个独立的 `OS` 对待，要求容器的 IP 地址与主机在同一个网段，如同 `OpenStack` 等云平台 `IaaS` 一样，动态地分配 IP 地址，在这里我

们要重点讲述几个用于虚拟网卡通信的概念。

### 1. veth pair

要理解 Docker 的默认 bridge 模式，则要先从 veth pair 开始。容器拥有自己独立的 Network Namespace 空间，所以在两个独立的网络空间中运行的进程就需要 veth pair 的帮助来通信。Docker 的 bridge 模式在主机网络空间与容器网络空间建立了一对 veth pair，这样容器可以与主机空间进行通信。如图 7-4 所示。

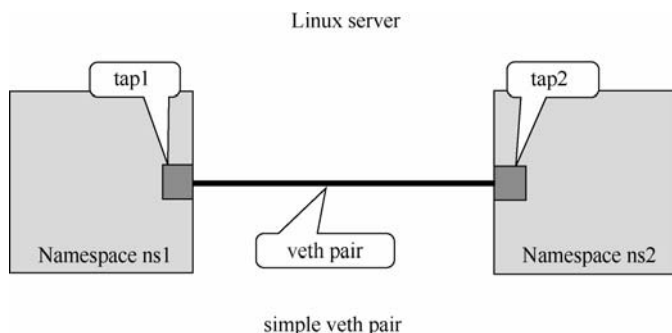


图 7-4 容器之间简单的 veth pair 通信

在一个主机上会有多个容器，为了让容器也能够互相通信，并且以一种中心相连的方式接入，这里引入了另一个我们需要理解的概念：bridge（网桥）。bridge 的作用如同一个虚拟的交换机，当出现多组 veth pair 时，将所有 veth pair 的一段加入 bridge 中，则所有地址空间的容器就可以相互通信了，加入网桥之后的通信图如图 7-5 所示。

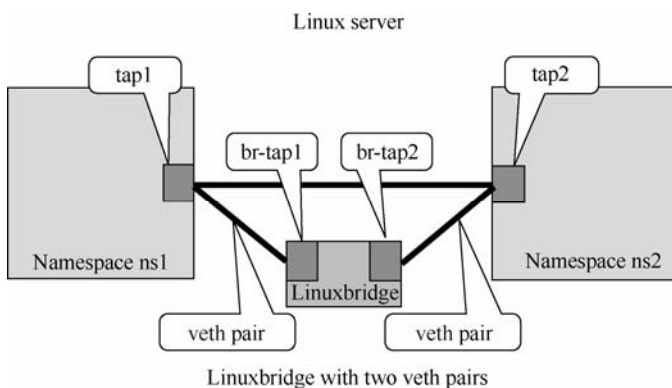


图 7-5 容器之间通过 Linuxbridge 通信

Docker 本身的网络解决方案比较简单，它开放了足够灵活的接口让用户来设置网络，很多开源项目对 Docker 的网络部分进行了补充，其中包括 pipework、weave、flannel 等，在 PaaS 平台上我们使用 pipework 来配置容器网络。

## 2. pipework

`pipework` 是一个 200 多行的 Shell 脚本，它使用了 `Network Namespace`、`veth pair` 及 `bridge` 来完成容器网络的设置。如果我们需要将容器 IP 地址配置成与主机 IP 地址在同一个网段，那么常规的步骤是：

- (1) 建立一个网桥；
- (2) 将主机网卡加入这个网桥；
- (3) 在主机上的网桥中配置 IP 地址，并将路由等信息设置为与此网桥相关；
- (4) 启动容器，配置一对 `veth pair`，将 `veth pair` 的两端分别加入容器及主机网桥中。

建立名称为 `public-bridge` 的网桥，将主机网卡 `eth0` 加入，并在主机上设置 IP 地址、路由等网络信息：

```
root@ubuntu:~# brctl addbr public-bridge
root@ubuntu:~# brctl addif public-bridge eth0
root@ubuntu:~# ip addr add 192.168.121.128/24 dev public-bridge
root@ubuntu:~# ip route add default dev public-bridge
```

启动名为 `test` 的容器，在这里将 `net` 模式设置为 `none`：

```
root@ubuntu:~# docker run -itd --name test --net=none ubuntu /bin/bash
```

安装 `pipework`：

```
git clone https://github.com/jpetazzo/pipework
cp ~/pipework/pipework /usr/local/bin/
```

使用 `pipework` 将 `test` 容器的网络地址设置为主机网段：

```
root@ubuntu:~# pipework public-bridge test 192.168.121.120/24@192.168.121.200
```

需要注意的是，在桌面虚拟机环境运行时因为虚拟机厂商对虚拟网卡的设置有其他要求，所以需要依据不同的虚拟化软件进行特别设置。而在生产环境中，我们将容器直接运行在物理操作系统上，既可以使用 `DHCP` 进行分配，也可以通过 `API` 服务器提供接口自行对 IP 网段进行维护。

## 7.4.3 容器链接与数据卷

容器的网络管理让我们将 `container` 服务以可选的模式提供给外部使用。对于一个完整的应用而言，为了组成一个完整的服务，它的所有组件并不能够打包成一个独立的容器计算单元，而是由多个容器组合而成的，例如前端使用 `Nginx` 服务器做代理，后端使用 `Tomcat` 做应用服务器，最后将数据写入到 `MySQL` 中，而 `MySQL` 的数据持久化地存储在一个独立的数据卷中。在这个组合中我们一共使用了 4 个容器。本节将介绍容器的链接与数据卷。

## 1. 容器链接

同一台主机上的两个容器除了使用网络端口映射的方式互相交互，还有一种更加轻量级的 Link 链接方式。我们先启动数据库中的一个 MySQL 数据库容器，之后启动引用它的 Tomcat 容器，并将 MySQL 容器链接在其上：

```
root@ubuntu:~# docker run -dit --name=mysql -e MYSQL_ROOT_PASSWORD=
1234 mysql
root@ubuntu:~# docker run -dit --link=mysql:database --name=tomcat_
app tomcat /bin/bash
49705b6a2cdf2481c6514518c6d4b5ec528257aa46e32d596a5eeef157e9926e
root@ubuntu:~# docker attach 49
root@49705b6a2cdf:/#
root@49705b6a2cdf:/# cat /etc/hosts |grep database
172.17.0.5      database
```

MySQL 对外提供服务的端口是 3306，由于没有选择 `--publish` 选项对外暴露端口，所以 Tomcat 容器使用了 link 的方式访问它。我们看到在 Tomcat 容器的 `host` 文件中自动加入了一条域名解析信息，将 link 的 `database` 解析成了 MySQL 数据库的私网 IP 地址 172.17.0.5，这也是 Tomcat 容器可以不通过随机 IP 地址访问 MySQL 数据库的关键所在：

```
root@49705b6a2cdf:/# telnet database 3306
Trying 172.17.0.5...
Connected to database.
Escape character is '^['.
```

在 Tomcat 容器内，telnet 与 MySQL 容器数据库端口 3306 是相通的，可在主机外部查看 iptable 策略。在 Forward 中加入了这两个容器之间通信的策略：

```
root@ubuntu:~# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source                destination

Chain FORWARD (policy ACCEPT)
target     prot opt source                destination
ACCEPT    tcp  --  172.17.0.5             172.17.0.10            tcp
spt:mysql ACCEPT    tcp  --  172.17.0.10            172.17.0.5             tcp
dpt:mysql ACCEPT    all  --  anywhere               anywhere               ctstate
RELATED,ES TABLISHED
ACCEPT    all  --  anywhere               anywhere
ACCEPT    all  --  anywhere               anywhere
```

以上是在同一个主机之间的两个容器通过 Link 方式进行通信的示例，其相对于网络端口映射的优势在于无须每次检查容器的随机 IP 地址。跨主机之间的 Link 通信方式常使用 ambassador 来完成，这种方式在主机间启用了两个相当于代理的容器来用于数据转发，如图 7-6 所示。

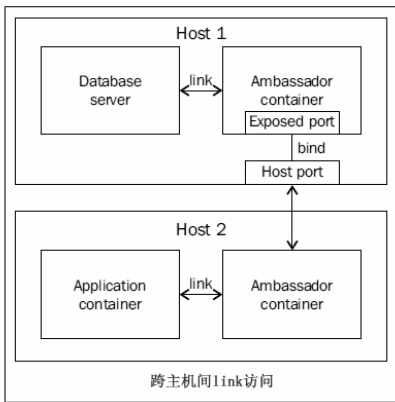


图 7-6 跨主机间的 link 访问

当 Host 1 上的数据库服务器重启后，只需要启动 Host 1 的 Ambassador，不需要重启 Host 2 上的应用服务器。

2. 数据卷

在讲解 run 命令时详细讲述了--v 选项的作用是将本地卷 mount 到容器中。在这里介绍另外一种方式，将一个独立的容器作为数据卷使用：我们启动一个 Ubuntu，并设置一个 MySQL 数据卷，之后在启动 MySQL 时将这个卷使用--volumes-from 挂载到容器上，这个 MySQL 的数据将写入 volume 容器中：

```
root@ubuntu:~# docker run -v /data/mysql --name data-volume ubuntu
echo "This is a data volume "
root@ubuntu:~# docker run -dit --name=mysql --volumes-from data-volume
mysql
```

这种方式用以防止 MySQL 容器 crash 后数据丢失的情况出现，下面启动一个数据备份容器：

```
root@ubuntu:~# docker run -d --volumes-from data-volume --name mysql-
backup \
-v $(pwd):/backup mysql $(mkdir -p /backup && cp -r /data/mysql
/backup)
```

数据链接与数据卷的容器间的交互方式在应用组合间可以使用到，在 PaaS 平台中通用的方式是通过 pipework 设置大网段的 IP 地址，并依据用户的选择，将分布式存储的卷 mount 到容器上。

7.5 Dockerfile

前面的内容帮助我们将 Docker 作为独立的逻辑计算单元放到 PaaS 平台中， PaaS 平台

中的镜像客户化是基于 base image 的，通过在容器中修改，commit 命令生成镜像是随后 push 到仓库中完成的，整个工作流程的控制由 PaaS 后台完成。本节将介绍 Dockerfile 的使用，它是另外一种生成客户化镜像的方式，它与 commit 命令的区别在于其全部的客户化过程都是默认安装的，无须人工交互，这对专业的系统管理人员、运维人员制作标准的中间件镜像、数据库镜像等非常有帮助。

### 7.5.1 基本指令集

让我们从一个最简单的实例开始，在本地创建一个 Dockerfile 文件，内容如下：

```
#选择 base 镜像
FROM ubuntu:latest

#添加作者信息
MAINTAINER yuhe002 "yuhe002@pignan.com.cn"

#安装 ssh package，并将监听端口修改为 2222
RUN apt-get install --force-yes -y ssh
RUN mkdir /var/run/SSHd
RUN sed -i 's/Port 22/Port 2222/g' /etc/SSH/sshd_config

#对外暴露 2222 端口
EXPOSE 2222

#将默认的命令设置为启动 SSHd 服务
CMD [ "/usr/sbin/sshd" , "-D" ]
```

随后我们通过 build 命令从本地读取 Dockerfile 文件，生成一个镜像 ubuntu-SSHd，完成后启动一个容器：

```
root@ubuntu:~ # docker build -t ubuntu-SSHd .
root@ubuntu: ~ # docker run -dit ubuntu-SSHd
```

通过 inspect 命令查看刚刚启动的容器的 IP 地址：

```
root@ubuntu:~ # docker inspect c667c2e35cdd |grep "IPAddress"
    "IPAddress" : "172.17.0.32" ,
```

在主机上使用 2222 端口 SSH 连接到容器中：

```
root@ubuntu:~# SSH root@172.17.0.32 -p 2222
root@172.17.0.32's password:
```

上面的 Dockerfile 示例文件在 Ubuntu base 镜像的基础上生成了一个有 SSHd 服务的新镜像，并且我们可以通过指定的端口 SSH 进入容器。

#### 1) FROM 指令

该指令用于指定基础镜像。



2) MAINTAINER 指令

该指令用于设定镜像制作者的信息。

3) RUN 指令

RUN 指令与 CMD 指令很容易让人误解，RUN 指令运行后所有的内容都将被持久化，例如安装 package、修改文件等，它是为了修改镜像数据而存在的，进程执行完毕后就会退出，不会保留在启动的容器中。

4) CMD 指令

CMD 指令是镜像设置的默认命令，它与运行的进程相关。

5) EXPOSE 指令

EXPOSE 指令告诉容器在启动时将哪些端口暴露给外部。这里需要注意的是，即便在 Dockerfile 中设置了该选项，在 run 命令执行时依然要设置-p 选项的映射关系。

以上是最基本的 5 条指令，在这个过程中我们使用了 build 命令进行镜像的生成。

build 命令的使用方式（见表 7-5）如下：

```
root@ubuntu:~ # docker build [OPTIONS] PATH | URL | -
```

表 7-5 build 命令所涉及的选项和说明

选 项	说 明
-t, --tag= " "	在镜像成功生成后，应用到镜像上的名称及标签
-q, --quiet	构建过程不显示过程信息，默认是显示的
--rm=true	成功构建后删除中间所有的临时容器
--force-rm	强制删除所有中间容器的数据，即便没有成功
--no-cache	在构建过程中不使用缓存

PATH、URL 指定了 Dockerfile 所在的位置，它会将这个目录下的所有文件传送到 Docker daemon 中。

7.5.2 环境指令集

1) ENV 指令

ENV 指令用来设置环境变量：

```
ENV <key> <value>
```

该指令将<value>的值设置到<key>中，随后的 Dockerfile 的所有 RUN 命令可以读取环境变量值。

## 2) WORKDIR 指令

该指令设置当前环境的工作目录，主要用来为 RUN、CMD 命令查找可执行文件的目录。指令可以在 Dockerfile 文件中多次使用，也支持使用相对路径。

## 3) USER 指令

USER 指令用来设置在 Dockerfile 脚本中运行 RUN 进程的用户，可以是用户名，也可以是 UID。

# 7.5.3 数据指令集

## 1) VOLUME 指令

VOLUME 指令和 RUN 命令中的 -v 选项类似，在容器中创建一个 mount 点，从外部挂载一个卷，可以是外部存储或者其他容器。

## 2) ADD 指令

ADD 指令用来将本地文件复制到镜像中：

```
ADD <src> <dest>
```

ADD 指令中的 <src> 所指的文件或者目录必须是相对于 build 命令的 PATH 或者 URL 路径的，<dest> 是镜像中的绝对路径，文件将复制到镜像中的这个位置。

## 3) COPY 指令

COPY 指令和 ADD 基本一样，最大的区别在于 COPY 只支持在本地文件中复制，也就是说如果你需要从 URL 外部或者 stdin 标准输入中复制文件，那么将不受 COPY 指令支持。

# 7.5.4 ENTRYPOINT 指令

RUN、CMD 和 ENTRYPOINT 这三条指令比较容易混淆，如前所述，RUN 命令用于在基础镜像上修改文件数据，将内容持久化而构建新镜像。在 Dockerfile 中有多条 RUN，在执行完成后任务便退出，而 CMD 和 ENTRYPOINT 只有一条。

CMD 是镜像生成容器后默认启动的命令。ENTRYPOINT 与 CMD 类似，也是容器默认执行的命令，但 ENTRYPOINT 让这个镜像更像一个独立的可执行文件，而在 RUN 命令之后输入的内容仅作为参数传入，我们通过一个例子来理解。

首先使用 CMD 作为最后默认的命令，在这里我们仅仅放一个 echo 命令：

```
CMD [ " echo " ]
```

随后运行一个容器：

```
root@ubuntu:~# docker run -it ubuntu-echo echo aa
```

它的输出是：

```
aa
```

即在 run 容器时，后面的 echo aa 覆盖了 Dockfile 中最后一行的 echo 命令。

如果我们使用 ENTRYPOINT 结尾，则：

```
ENTRYPOINT [ "echo " ]
```

同样，我们运行一个容器：

```
root@ubuntu:~# docker run -it ubuntu-entry-echo echo aa
```

那么它的输出是：

```
echo aa
```

从这里可以看到“echo aa”在 RUN 命令中不是作为 CMD 传入覆盖的，而是作为参数传给容器执行的，而这个容器相当于一个 echo 可执行文件。

# 分布式协调 ZooKeeper

## 8.1 ZooKeeper介绍

PaaS 平台的核心是资源管理与任务调度，它们负责数据中心网络、存储、计算资源的分配与管理。而在核心的背后还有一个“影子”模块支撑起了整个平台的协作与通信，它要承担全局性锁的职责，同时要将平台的非业务性元数据保存起来，动态地通知到订阅者，在分布式情况下需要一个控制多任务间的同步方法，如同现实世界中的交通灯一样，协调、管理着从各个路口汇聚的车辆。当我们在各种分布式应用中将这功能独立出来时，会逐渐形成一个专门的功能领域——分布式协调管理，本章将介绍 Hadoop 家族的 ZooKeeper，其以一种非学院派的简捷方式完成了复杂的分布式协调工作。

### 8.1.1 ZooKeeper是什么

如果你有并发任务的编程经验，那么我想你肯定会遇到这样一个场景：两个或多个并发任务之间如何共享数据，如何进行通信。Linux 的多进程编程有多种 IPC 方法提供我们使用，包括管道、消息队列、共享内存，以及域套接字等，我们利用这些编程接口实现了全局变量，完成了进程间的通信，同时保证了性能的高效。如果我们将这个单节点操作系统的多任务搬到分布式场景下，在这一场景下由于引入了网络这个通信的桥梁，可用性、可靠性与单机相比变得不再不可控。当一条消息因为网络故障而丢失时，消息的接收方与发送方以怎样的方式来处理这种“故障”，才能尽可能规避这种网络异常，ZooKeeper 能够规避这种网络故障吗？答案是否定的，我们不能期望分布式系统中的组件能解决分布式系统自身的问题，那 ZooKeeper 又能做什么呢？在良好的物理位置部署条件下，ZooKeeper 能够最小化地忽略网络故障，完成协调管理工作。

ZooKeeper 非常简单，这种简单是指非常容易上手，设计上的清晰与简捷性能够让用户轻松地将 ZooKeeper 集成到分布式环境中。ZooKeeper 的核心是一个类似于树形结构的文件系统，提供了一组精简的操作原语与功能。ZooKeeper 的这组精简操作使用户能够自行组

合，适应分布式环境下的多种场景，包括分布式锁、分布式消息队列、主节点选举、动态配置服务等。

让我们一起来看看那些使用 ZooKeeper 的著名开源项目，就可以知道这些 ZooKeeper 的适用性有多么广泛。

- **Aapach Mesos:** Mesos 资源管理器使用 ZooKeeper 作为其高可用的保障，它通过 ZooKeeper 在多个 Mesos 管理节点中选举出一个 Master，并通过 ZooKeeper 完成健康侦测功能。
- **Aapach Kafka:** Kafka 是一个订阅-发布的分布式消息队列系统，它使用 ZooKeeper 做健康检查，并且通过 ZooKeeper 来实现全局的消息话题（Topic）功能，保证在集群场景下生产者、消费者对话题订阅的状态保持一致。
- **Apache Solr Cloud:** ZooKeeper 的另一个显著功能是保存分布式系统中的元数据，Solr Cloud 是一个企业级的分布式搜索平台，它基于 iucene 实现，在分布式场景中使用 ZooKeeper 保存集群的元数据。

ZooKeeper 的适用性在于它绝不做多余的工作，它提供选举功能的接口、集群健康探测接口，而不是代替应用程序去实现这些功能。ZooKeeper 也不是一个海量的存储系统，而是仅负责处理应用的分布式元数据信息。

### 8.1.2 ZooKeeper架构

虽然 ZooKeeper 是为了解决分布式系统中的协调问题而存在的，但其本身依然是分布式系统内的应用程序。在体系结构上，ZooKeeper 采用两种模型的混合式：Client-Server 模型用于对分布式协调系统中的对象提供协调服务，Peer-to-Peer 模型用于在 ZooKeeper 集群内交互数据。这种混合的体系结构让 ZooKeeper 既保持客户端的简单化，又能够在集群内部保持一个良好的高可用机制。

ZooKeeper 有两种运行模式：standalone 和 quorum。standalone 代表着 ZooKeeper 由一个单一节点构成，它的数据并不需要在多节点之间复制；quorum 则多用于生产环境，ZooKeeper 由多个节点构成一个集群，quorum 的含义是“法定人数”，这个名称与 ZooKeeper 的 Leader 选举、集群节点间的数据同步有关。

当我们采用 quorum 模式时，集群中的所有节点启动时会发起一次选举，每个节点都会进行投票，投票数超过且最多“法定人数”的节点将成为 Leader，我们会在 8.13 节详细地介绍选举。对于用户的读操作，ZooKeeper 集群中的所有节点将对其接收处理；而对于写操作，ZooKeeper 将它作为事务来处理，并且只有 Leader 才能发起事务，在每次事务提交之前，修改的数据要在集群中同步。我们可以看到在读操作上 ZooKeeper 可以具备很高的伸缩性，而当遇到写时，ZooKeeper 选择了一种集中的处理方式，让问题变得更加简单。quorum

在数据同步时又一次发挥了作用，它不仅仅代表着选举 Leader 时通过的票数，还代表着修改一次事务的数据时，在同步多少个节点后即可将成功消息反馈给客户端，而不必等到集群中的所有节点全部完成同步。

quorum 模式对于“法定人数”的选择非常重要，我们假设在一个 5 个节点的 ZooKeeper 集群中，quorum 的选择是 3，当 3 个节点的事务数据同步完成后，客户端将会收到成功反馈，剩余 2 个节点可以随后做数据同步。在分布式环境中，消息延迟、系统崩溃的情况时有发生，而这个 quorum 值要在这种“恶劣”条件下使数据保持准确性。

如图 8-1 所示是 ZooKeeper 的架构模型。

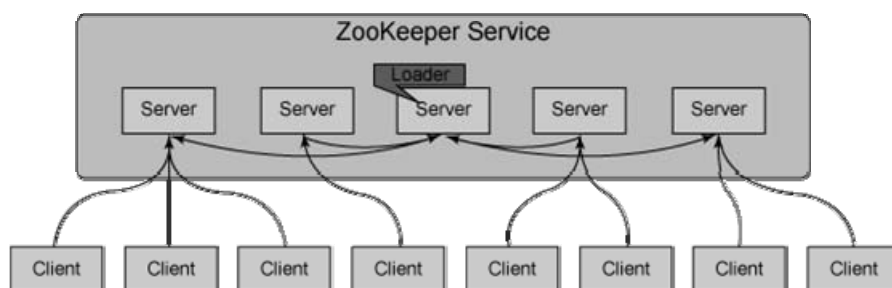


图 8-1 ZooKeeper 的架构模型

再回到 5 个节点的 ZooKeeper 集群，我们将 quorum 设置为 2，Leader 和其中一个 Follower\_1 同步了数据，满足事务提交的条件，将结果返回给客户端。这时假设 Leader、Follower\_1 所在的网络出现了问题，与其他 3 个节点 Follower\_2、Follower\_3、Follower\_4 及客户端失去联系，但修改后的数据还没有同步到这 3 个节点上，这时通过重新选举 3 个节点出现了一个新的 Leader，在随后的对同一数据事务的修改中，这三个节点中没有一个节点上有上一个 Leader、Follower\_1 同步了的数据内容，于是事务再一次提交，对于同一个数据出现了不一致的值，等 Leader、Follower\_1 从灾难中恢复过来时，谁也不知道哪个数据是准确的。这种情形在分布式中有一个专门的名称——脑裂（Split-Brain），当一个整体一分为二、自成一体、没有限制时，它们就可以产生不一致的数据了。这里的问题就在于对 quorum 的选择，对于每一次事务的提交都需要一定数量的集合成员同意，我们要保证的是在每次参与“讨论”的成员里总有一个是持有最新数据的，这样才能在左右脑间牵上一条线。很显然，quorum 最小值应该是节点总数的一半再加 1。关于 quorum 的设置，ZooKeeper 参照人类的现实行政管理结构，在节点上允许设置不同的权重，权重越高的节点投票分量越重，如同美国两会参议院、众议院议员在国会议案的投票占比一样。

### 8.1.3 数据模型

ZooKeeper 通过集中型的 Client-Server 架构服务于客户端，采用点对点的 Pair-Pair 架构

在各节点间同步数据，ZooKeeper 内部的数据结构和数据类型采用了精简的方式提供给用户使用。

在数据结构上，ZooKeeper 采用了类似文件系统的树形结构，树中的节点被称为 znode，在整棵树中有着清晰的父子层级关系。我们通过图来解析 ZooKeeper 树形结构，在这个服务中实现的是一个工厂中工作任务的分派，ZooKeeper 在 root 下有 4 个 znode，其中：路径是 /factory，代表所在工厂；/workers 代表工厂中所有的工人数，子节点 /workers/woker-1 代表一个工人“张三”；/tasks 节点代表工厂中所有的任务数；/tasks/taks-1 代表有一项盖房子的任务；最后一个节点体现了 worker-task 这种模型的分派关系；/assign 下有着工人与任务的对应关系。如图 8-2 所示。

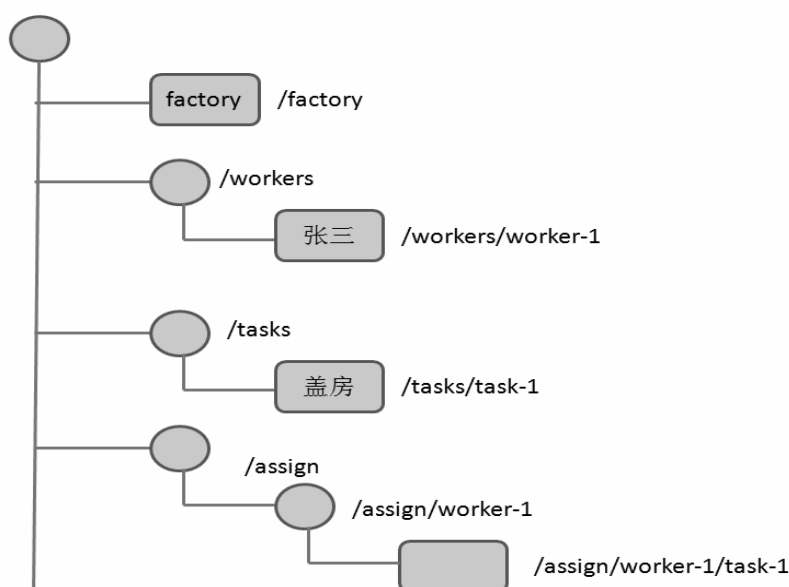


图 8-2 工厂任务分配关系

ZooKeeper 对数据的修改都当作一次事务来看待，每一次的事务动作都是原子操作，并分配一个唯一的 zxid (zookeeper transaction id)，在每一个 znode 中都记录了与之相关的 zxid。znode 除了有效载荷数据，还包括如下属性。

- czxid: 引发当前 znode 改变的 zxid。
- mzxid: 当前 znode 上一次修改前的 zxid。
- ctime: 对 znode 事务操作所花费的时间，单位是毫秒。
- mtime: 操作上一次 znode 事务所花费的时间，单位是毫秒。
- version: 除了关联到 zxid，znode 自身有一个版本属性，表示当前 znode 修改的次数。

- **cversion**: znode 子节点的修改次数。
- **aversion**: znode ACL（访问列表）的修改测试，znode 可以通过某些规则对其进行权限控制。
- **ephemeralOwner**: 后面我们会谈到短暂型的 znode，它的存活性与创建它的客户端会话保持一致，记录了客户端会话 ID，如果是持久型 znode，那么该项为 0。
- **dataLength**: znode 数据字段的长度。
- **numChildren**: znode 子节点的个数。

### 1. 持久与短暂 znode

znode 在创建时可以指定模式，我们可以用不同的模式组合成需要的分布式场景功能。一个 znode 节点可以是持久的（Persistent），也可以是短暂的（Ephemera）。一个持久的 znode 只有在收到删除命令后才能被删除，短暂的 znode 则会在客户端与服务端 ZooKeeper 之间保持的会话中断时自动删除，这可能是客户端正常退出，也可能是异常宕机。在客户端与服务端之间一旦建立连接，则会有定期的心跳检查以保证这个长连接的存活。

持久的 znode 用于保存结构信息，例如图 8-2 中的 /workers、/tasks 节点，不管创建它的客户端是否还保持着会话，这些结构性节点不应当随着创建者的离开而消失。短暂的 znode 则体现了其动态的一方面，它随着客户端的会话中断而消失，例如图 8-2 中具体的工作者“张三”，它可能是一个工作进程，在空闲时主动注册到 ZooKeeper，在认领到任务后断开连接，自动删除 /workers/work-1。一个暂时性的 znode 除了会话结束自动删除，也能够像持久 znode 一样被命令删除。

### 2. 序列 znode

序列 znode 是指在创建时自动分配一个唯一的递增的整数，这个整数序列可以作为路径名称的一部分。例如，客户端创建了一个序列 znode，为 /worker/work-，那么 ZooKeeper 会自动分配一个序列数，如果是当前 znode 下的第一个子节点，那么分配 1，整个路径就变成了 /worker/work-1。序列节点的最大作用是生成唯一名称节点，同时能提供节点间的创建次序信息。

### 3. ACL

在 znode 的属性中还有一项专门用于用户的权限控制——ACL，它可依据一些条件对 znode 的某些操作进行控制。ZooKeeper 的 ACL 也建立在认证与授权机制下，它包含了以下认证方式。

- （1）**digest**: 通过用户名/密码对客户端进行认证。
- （2）**sasl**: 支持客户端通过 Kerberos 认证。



(3) ip: 通过 IP 地址对客户端进行认证。

在授权方面, ACL 可以对 `znode` 的创建(子节点)、读取、写入、删除及管理(设置 ACL)进行控制。

### 8.1.4 监听与通知

ZooKeeper 的主要功能是分布式协调, 在大环境中共享数据信息, 数据的变化对于关注它的客户端来说非常重要。对数据的变化越及时地通知到客户端, 分布式系统的流通效率就越高。假如客户端为了获取一个数值的变化情况, 不停地向服务端发送 `getDate` 请求, 则这对于分布式的远程调用方式资源消耗是很昂贵的, 同时对服务器也会造成很大的负载。然而对于一个不经常变化的数值来说, `get` 回来的值是没有变化的, 这些 `get` 动作是完全没有必要的。

在消息队列中我们通常使用一种发布-订阅的消息传递机制, 这在 ZooKeeper 中同样适用, 因为客户端会与 ZooKeeper 保持一个基于 TCP 的长连接会话, 对于一个值得客户端关注的数值, 在第一次发送请求到服务端后就可以监听(Watch)到其上, 当它发生变化时服务端会主动通知(Notify)客户端, 触发客户端的事件(Event)。在这种通知机制中要特别注意的是, 在一个数值上发生的两个变化引发了两次通知行为, 这两次行为必须是保持次序的。

在 ZooKeeper 中有不同类型的监听方式, 它可以依据 `znode` 数据值的变化设置监听, 也可以依据 `znode` 删除、新增设置, 还可以依据 `znode` 下的子节点变化设置。这种设置是在 ZooKeeper 客户端向服务端发起请求时设置的, ZooKeeper 有多种开发语言的 API 库。

### 8.1.5 API集合

ZooKeeper 可以满足分布式系统的各种协调管理需求, 如下所述。

- 名称服务: 名称服务是一种映射服务, 将一类名称映射为用户更需要的信息。DNS 服务器将域名映射为 IP 地址, 从而可以在互联网上路由通信, 电话簿将人名映射为电话号码, 这样我们就可以拨打其中的号码。在分布式系统中, 我们很可能要将某些系统的访问信息放在共享处以供其他节点访问。
- 锁定: 为了允许在分布式系统中对共享资源进行有序访问, 保护共享资源的原子性, 以及不被并发访问所毁坏, 可能需要实现分布式互斥(Distributed Mutexes)。
- 同步: 与互斥同时出现的是同步访问共享资源的需求。这是一组规范的计算机同步控制原语, 可以实现为一个生产者-消费者队列模型, 还可以是一组信号控制, 在 ZooKeeper 中形象地将这种信号控制称为“障碍”(Barriers)。

- **配置管理：**ZooKeeper 可以用来集中存储和管理分布式系统的配置。这意味着，所有新加入的节点都可以在加入系统后立即使用来自 ZooKeeper 的最新集中式配置。这还允许你通过其中一个 ZooKeeper 客户端更改集中式配置，集中地更改分布式系统的状态。
- **领导者选举：**分布式系统可能必须处理节点停机的问题，将短暂类型的 `znode` 作为 Leader 节点的健康检查，一旦会话丢失则发起选举动作。

就以上分布式需求，ZooKeeper 并不提供最直接的原语，如果要支持所有的分布式场景，就不得不设计出所有的操作原语，例如锁控制有 `lock`、`release` 等，随着未来新的需求，这些操作原语还要随之变化。ZooKeeper 采用了一种提供底层基础 API 的方式，让用户自行实现各类分布式需求，这类 API 是一组类似于文件访问的简单操作，如下所述。

- `create /path data`：用 `/path` 名称创建一个 `znode`，同时包含数据 `data`。
- `delete /path`：删除一个名称为 `/path` 的 `znode`。
- `exists /path`：检查名称为 `/path` 的 `znode` 是否存在。
- `setData, getData /path data`：设置或返回名称为 `/path` 的 `znode` 数据为 `data`。
- `getChildren /path`：返回 `/path` 下的所有子节点列表。
- `getACL, setACL /path`：设置或返回 `/path` `znode` 的权限列表。

在服务端中传递的所有数据都是二进制数据，ZooKeeper 本身并不会对数据进行字符集转换或者其他解析，这些数据的序列化工作由外部库来完成。每次设置或读取一个 `znode` 的数值，这个动作都是一次性完成的，是一个原子动作，不会被其他事项打断，而会出现一个部分写入的状态。

### 8.1.6 会话

接下来讨论 ZooKeeper 会话（`session`）。会话是与状态有关联的。为了说明状态的重要性，我们用小旅馆和高级酒店来打比方：对于小旅馆，它从来不记录用户的入住历史信息，对于每一个用户的每一次入住都提供同样的服务；而高级酒店为了提升服务质量，会将用户的历史信息记录下来，在下次用户入住时会根据这些信息提供更加专业的服务，它们与用户建立了一种“会话”关系，这种关系是有状态的，随着用户的喜好而变化。HTTP 在设计之初是无状态的，因为它仅提供静态的访问页面，完成链接关系，但随着电子商务的发展，服务器必须对一个用户的两次访问进行记录，于是服务器使用了 HTTP header 中的 `Cookies` 自动标明一个用户，实现状态控制。

ZooKeeper 与客户端建立好连接后也会产生一个会话，这个会话的超时时间是由客户端应用程序决定的。如果在这段时间内 ZooKeeper 服务端与用户一直是空闲状态，没有接收到

任何请求，则会话超期，ZooKeeper 上保存的与此会话相关的所有短暂 znode 将被全部删除。

客户端与 ZooKeeper 会有一种心跳机制，它定期向服务端发起 ping 请求，心跳机制的频率要短于会话超期时间，这样才可以在会话超时前检测服务器是否异常，如果有异常，那么客户端会自动重新连接到另一台正常的服务器。值得注意的是客户端之前保持的会话在新的服务上依旧有效，由此可以看出 ZooKeeper 的会话在集群内是全局性的。

### 8.1.7 观察者

在 ZooKeeper 集群中除了 Leader、Follower，还有第三类角色，即观察者（Observer）。引入观察者的主要目的是提升集群的整体性能，同时不增加选举及数据同步时投票的复杂性。另外，参与投票的成员越多，在数据同步上所消耗的时间越长。我们可以建立这样一个集群环境：以 5 个节点作为 Leader、Follower 存在，将 quorum 设置为 3，投票数超过 3 时认为数据修改成功。在这样的一个集群中有 3 个节点的数据一定是最新的，而其他 2 个节点在随后进行数据同步。在 5 个节点外我们加入集群的其他节点全部以 observer 角色身份存在，其主要目的是保证分布式系统中的性能。这时又有另一个问题出现了，虽然缩短了数据事务提交的时间（只需要 3 个投票者），但访问最新数据的概率要大大降低（所有 observer 都是随后同步数据的），因此对于一致性要求较高的 znode，客户端在 get 数据时可以强制要求同步，即 Leader 获取最新数据。

## 8.2 ZooKeeper使用

### 8.2.1 快速安装

ZooKeeper 的安装非常简单，因为它是采用 Java 语言编写的，在运行它之前需要安装 Java 虚拟机，在这里我们部署一个由三个 ZooKeeper 节点组成的集群。在写作本书时，最新的 ZooKeeper 版本是 3.5.0。我们将三个节点放在三台 Ubuntu 操作系统上，它们的 IP 地址分别是 192.168.121.1、192.168.121.2、192.168.121.3，下面我们开始对每台服务器进行安装。

安装 ZooKeeper 所依赖的 Java 运行环境：

```
yuhe@ubuntu:~$ apt-get install jre-default
```

下载 ZooKeeper 3.5.0 的压缩包，建立一个 ZooKeeper 目录，将其解压：

```
yuhe@ubuntu:~$ wget http://apache.fayea.com/zookeeper/zookeeper-3.5.0-alpha/zookeeper-3.5.0-alpha.tar.gz
```

## PaaS 实现与运维管理：基于 Mesos + Docker + ELK 的实战指南

```
yuhe@ubuntu:~# mkdir /app && cp zookeeper-3.5.0-alpha.tar.gz /app
yuhe@ubuntu:~# cd /app && tar zxvf zookeeper-3.5.0-alpha.tar.gz
yuhe@ubuntu:~# ln -s zookeeper-3.5.0-alpha zookeeper
```

创建一个目录，用它来存储与 ZooKeeper 服务器有关联的数据，在目录中创建一个 myid 文件，写入 ID 号，1 代表集群中的第一台服务器，其他两台类同：

```
yuhe@ubuntu:~# mkdir -p /var/zookeeper/data
yuhe@ubuntu:~# echo "1" /var/zookeeper/data/myid
```

创建或编辑 zookeeper/conf/zoo.cfg 文件，在启动 ZooKeeper 时需要用到：

```
yuhe@ubuntu:~# cp zoo_sample.cfg zoo.cfg
```

文件内容如下：

```
tickTime=2000
initLimit=10
syncLimit=5
dataDir=/var/zookeeper/data
clientPort=2181
server.1=192.168.121.1:2888:3888
server.2=192.168.121.2:2888:3888
server.3=192.168.121.3:2888:3888
```

在配置文件中有三个端口号需要说明，2181 是专门向客户端提供服务的端口，2888 是集群内各节点进行数据交互的端口，而 3888 是进行 Leader 选举时用到的。在配置文件中标明了有 3 台服务器，在 server 后面的数字是服务器序号。

- **tickTime**：这个时间设置很重要，前面也提到过，在 ZooKeeper 服务器之间或者服务器与客户端之间有一种心跳机制，用以检测对方的健康状态，这个时间是健康检查的频率，即每隔 tickTime 时间就会发送一个心跳，它的单位是毫秒，在这里设置的心跳时间为 2s。
- **dataDir**：ZooKeeper 所产生的数据将放到该目录下，在这里是我们之前为其创建的数据目录，目录下的 myid 文件要与集群的 server ID 对应。
- **clientPort**：该端口是与客户端交互时所使用的服务端口。
- **initLimit**：这个配置项代表着集群中的其他节点在与 Leader 建立连接，以及建立完成后同步数据所运行的最大时间，它是以 tickTime 的倍数计算的，在这里我们将时间长度设置为  $10 \times 2000 = 20$  秒。
- **syncLimit**：这个配置项表示集群中 Leader 与其他节点进行数据同步所运行的最大时间，它也是以 tickTime 的倍数计算的，在这里我们将时间长度设置为  $5 \times 2000 = 10$  秒。

在启动 ZooKeeper 之前我们要将 Ubuntu 上的 sh 命令指向修改为 bash。Ubuntu 从 6.10

版本开始，默认使用 dash (the Debian Almquist Shell) 而不是 bash (the GNU Bourne-Again Shell)，但这个版本的 Shell 在执行 ZooKeeper 的 zkEnv.sh 时会因为版本不兼容而报错，我们将其链接到 bash 上以解决：

```
yuhe@ubuntu:~# rm -rf /bin/sh
yuhe@ubuntu:~# ln /bin/bash /bin/sh
```

我们依次在三台服务器上完成以上配置后，执行 zkServer.sh 脚本启动服务：

```
yuhe@ubuntu:~# /zookeeper/bin# ./zkServer.sh start
./zkServer.sh start
JMX enabled by default
Using config: /app/zookeeper/bin/./conf/zoo.cfg
Starting zookeeper ... STARTED
```

使用 status 命令在三台机器上检查 ZooKeeper 的状态，我们看到机器的服务被选举成了 Leader：

```
yuhe@ubuntu_1:~ /zookeeper/bin# ./zkServer.sh status
JMX enabled by default
Using config: /app/zookeeper/conf/zoo.cfg
Client port found: 2181
Mode: leader
```

```
yuhe@ubuntu_2:~ /zookeeper/bin# ./zkServer.sh status
JMX enabled by default
Using config: /app/zookeeper/conf/zoo.cfg
Client port found: 2181
Mode: follower
```

```
yuhe@ubuntu_2:~ /zookeeper/bin# ./zkServer.sh status
JMX enabled by default
Using config: /app/zookeeper/conf/zoo.cfg
Client port found: 2181
Mode: follower
```

## 8.2.2 基本操作

现在，我们可以从其中一台正在运行 ZooKeeper 服务器的机器上启动一个 CLI 客户端。

```
yuhe@ubuntu_1:~ /zookeeper/bin# ./zkCli.sh
```

我们可以创建、编辑和删除 znode。让我们创建一个 /factory 路径的 znode，作为一个 helloworld 的示例。

创建 factory znode：

```
[zk: localhost:2181(CONNECTED) 1] create /factory helloworld
Created /factory
```

通过 `get -s` 获取 `znode` 数据，`-s` 选项会返回详细的元数据信息：

```
[zk: localhost:2181(CONNECTED) 2] get -s /factory
helloworld
cZxid = 0x100000018
ctime = Sat Jun 20 06:18:03 PDT 2015
mZxid = 0x100000018
mtime = Sat Jun 20 06:18:03 PDT 2015
pZxid = 0x100000018
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 10
numChildren = 0
```

使用 `delete` 删除 `factory` 节点：

```
[zk: localhost:2181(CONNECTED) 3] delete /factory
```

接下来让我们创建工作者/`worker`，并查看其数据信息，当前内容是 `stronger`：

```
[zk: localhost:2181(CONNECTED) 11] create /workers strong
[zk: localhost:2181(CONNECTED) 12] get -s /workers
stronger
cZxid = 0x10000001a
ctime = Sat Jun 20 06:20:04 PDT 2015
mZxid = 0x10000001a
mtime = Sat Jun 20 06:20:04 PDT 2015
pZxid = 0x10000001a
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 8
numChildren = 0
```

使用 `-w` 选项监听/`workers` 上的变化：

```
[zk: localhost:2181(CONNECTED) 13] get -w /workers
stronger
```

在另一台机器上启动客户端，修改/`workers` 的内容：

```
[zk: localhost:2181(CONNECTED) 0] set /workers weakness
```

我们可以看到在第一台机器上返回了通知信息：

```
[zk: localhost:2181(CONNECTED) 14]
WATCHER::
```

```
WatchedEvent state:SyncConnected type:NodeDataChanged path:/workers
```

通过 create 的 -s 选项创建序列 znode:

```
[zk: localhost:2181(CONNECTED) 6] create -s /workers/worker-
Created /workers/worker-0000000000
[zk: localhost:2181(CONNECTED) 7] create -s /workers/worker-
Created /workers/worker-0000000001
[zk: localhost:2181(CONNECTED) 8] create -s /workers/worker-
Created /workers/worker-0000000002
[zk: localhost:2181(CONNECTED) 9] create -s /workers/worker-
Created /workers/worker-0000000003
```

通过 stat 命令查看 workers 的状态:

```
[zk: localhost:2181(CONNECTED) 10] stat /workers
cZxid = 0x10000002a
ctime = Sat Jun 20 06:27:08 PDT 2015
mZxid = 0x10000002a
mtime = Sat Jun 20 06:27:08 PDT 2015
pZxid = 0x10000002e
cversion = 4
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 8
numChildren = 4
```

在上面的示例中, 我们使用的是 ZooKeeper 自带的 CLI 客户端与 ZooKeeper 服务器进行交互。ZooKeeper 到目前为止已经提供了 Java、C、Python 等多种语言客户端。我们可以使用这些 API 将分布式功能嵌入到自己的应用程序中。

## 8.2.3 配置参数

### 1. 基本配置

在 ZooKeeper 3.5.0 版本之后, ZooKeeper 新增了动态配置功能, 当我们将下面的配置放到静态文件中时, ZooKeeper 自动将一些参数移动到动态配置文件中, 我们在 8.2.4 节将介绍动态重配置。

下面讲解 Zookeeper 的基本配置。

#### 1) clientPort

为用来与客户端进行连接的 TCP 服务端口, 在默认情况下, ZooKeeper 将在主机所有网卡的地址上监听, 默认端口是 2181:

#### 2) lientPortAddress

为从 ZooKeeper 3.3.0 版本引入的参数, 指定面向客户端提供服务的地址, 可以是

IPV4、IPV6 或主机名，如果不设置，那么服务将绑定到所有地址的 `clientPort` 端口上。

### 3) dataDir

ZooKeeper 有两种存储文件，一种是快照，另一种是事务日志。在默认情况下这两个文件都放在 `dataDir` 目录中，在 `dataDir` 中需要有一个 `myid` 文件对应到 `zoo.cfg` 的 `server.id`。

ZooKeeper 在 `dataDir` 中的快照文件（`snapshot`）并不包括当前节点运行的所有数据，它的命名格式为 `snapshot.+最后一个更新的 zxid (snapshot.<zxid>)`。ZooKeeper 的每一次数据变化都会以事务日志的形式记录到磁盘中，而当日志条目达到一定随机数量时，ZooKeeper 会启动一个 `snapshot` 后台线程将当前事务日志及之前的内存数据全部导出到快照中，这个工作线程并不会对前端访问造成较大的性能影响或延时。`snapshot` 文件只记录了它导出时的所有数据，随后的事务操作并不在其中，如果需要恢复数据，那么需要将快照文件与事务日志结合在一起。

### 4) dataLogDir

如果没有设置 `dataLogDir`，那么事务日志文件将与快照放置在同一个目录中。事务日志的数据操作是同步的，必须将数据刷新到磁盘后，ZooKeeper 才认为事务完成。如果事务日志写入操作缓慢，那么将直接影响到 ZooKeeper 服务器的性能。事务日志操作对与它同在一个物理存储设备的其他活动非常敏感，不仅仅是快照的导出。在生产环境中为了保证服务器的高吞吐量，我们会将 `dataLogDir` 放在一个独享的专用存储设备上。

### 5) tickTime

`tickTime` 是 ZooKeeper 中用来衡量时间的基本单位，它代表了一个时钟滴答的长度，单位是毫秒。之前提到的会话超时时间、集群同步超时时间等都是基于这个 `tick` 而设置的。一个最小的会话超时时间是 2 个 `tick`，而集群最小的超时时间是 1 个 `tick`。`tick` 的默认设置是 3s。

## 2. 存储配置

ZooKeeper 服务端的数据存储在三个地方：内存数据库、全量快照及事务日志中。如前所述，在 ZooKeeper 中数据存储的结构是类似于文件系统的树形结构，如同一个内存数据库，保存了 `znode` 与会话信息。在 `dataDir` 中有一批 `snapshot` 快照文件，它依据一些条件定期从内存中刷新数据保存，而 ZooKeeper 中每一次数据的变化会以事务的形式将日志按顺序记录到存储设备中。

### 1) preAllocSize

ZooKeeper 中每一个日志文件的空间是提前分配的，默认是 64MB，可以通过 `preAllocSize` 来调整这个预分配值。`znode` 每一次的数据变化都会产生一条写操作到日志文件中，并且必须写操作完成后才返回给客户端事务，因此 ZooKeeper 要在设计上降低每一次



写损耗。设置预分配空间的目的是在写操作时减少对文件系统元数据的修改，从而提升性能。

每次生成一个快照后，ZooKeeper 将启用一个新的日志文件，如果前一个日志文件没有写满，则会造成磁盘空间的浪费，preAllocSize 的设置与 snapCount 相关。

## 2) snapCount

触发生成快照文件的条件在于当前已经积累了多少条事务日志，当事务条数大于  $(\text{snapCount}/2) + \text{Random}(\text{snapCount}/2)$  时生成快照。我们可以看到这并不是以 snapCount 为标准，而是选择一个随机值，这样做的目的是防止所有节点同时做快照生成动作。snapCount 的默认值是 100000。

## 3) autopurge.snapRetainCount

ZooKeeper 可以通过事务日志和快照文件还原内存中的数据信息，但在生成这些文件时会产生冗余信息，ZooKeeper 会定期对这些文件进行清理。autopurge.snapRetainCount 控制保留快照文件的数据，以及与其关联的事务日志文件。

## 4) autopurge.purgeInterval

快照文件、事务日志清理的时间频率的单位是小时，如果设置为 0，则不会自动启动清理程序，用户可通过 zkCleanup.sh 脚本清理。

## 5) fsync.warningthresholdms

ZooKeeper 完成一个事务时，将数据放到存储中所耗费时间若超过一个值则触发一个警告，单位是毫秒。

## 6) traceFile

相当于 Debug 输出，它会将 ZooKeeper 的每一个操作记录下来，由于你要把所有的选项写入到磁盘，所以打开该选项后会对性能造成影响，文件名是 trace-File.year.month.day。

## 7) syncEnabled

observers 角色的 ZooKeeper 节点默认同 Leader、Follower 一样，将事务日志同步写入磁盘。为了提升 observers 服务重启的效率，缩短启动时间，可以将此项设置为 false，默认是 true。

# 3. 网络配置

## 1) globalOutstandingLimit

ZooKeeper 接收来自客户端的请求时，会首先将请求保存在一个队列中，之后从队列中获取并处理。当来自于客户端的请求量巨大时，存放在队列中的请求量很容易将 ZooKeeper 的内存消耗殆尽，也就是常说的 out of memory。为此，我们可以设置一个保护机制，当一

个节点的队列中的请求量超过 `globalOutstandingLimit` 时，后续过来的请求将直接被丢弃，该项的默认值是 1000。

### 2) `maxClientCnxns`

除了放置在内存队列中的请求消耗资源，建立与保持一个网络连接也很消耗资源，如果不对客户端在连接的建立上加以限制，那么客户端可以向 ZooKeeper 发起大量的连接建立请求，当超过操作系统所能够承受的范围时，整个 ZooKeeper 服务都将不可用。

### 3) `minSessionTimeout`

虽然会话的超时时间是由客户端设置的，但在服务端上会设置一个最小会话超时时间，如果客户端设置的值小于它，那么以服务端设置的为准，其默认值为 2 个 `tickTime`。

### 4) `maxSessionTimeout`

如同 `minSessionTimeout`，服务端设置了一个最大会话超时时间，默认是 20 个 `tickTime`。

## 4. 集群配置

### 1) `initLimit`

如果超过 Follower 与 Leader 建立连接及同步数据所运行的最大时间，则认为访问 `maxClientCnxns` 用于针对用户 IP 地址限制连接建立数，默认值是 60。该选项是针对单台服务器的，如果集群中存在 5 个节点，则客户端最多可建立 300 个连接。如果 ZooKeeper 中保存的数据量大，网络带宽小，则传输数据会消耗一定的时间，这时有必要将此设置加大。

### 2) `syncLimit`

与 `initLimit` 类似，但此选项仅关注与数据同步。

### 3) `leaderServes`

指 Leader 是否接收客户端的请求，默认值是 “yes”。为了提升分布式环境下 ZooKeeper 的读操作吞吐量，可以关闭 Leader 对客户端提供服务，让其关注于协调功能。在集群中有足够多的 Follower、observer 节点时可以考虑关闭此选项。

### 4) `server.x=[hostname]:port:port[:observer]`

设置集群中的节点，Zookeeper 中的节点需要知道彼此的位置，在配置文件中我们可以按照这种格式设置集群服务器列表。`server.x` 中的 `x` 代表服务器 ID，其是一个整数，当服务器启动时查询数据目录中的 `myid` 文件，以此找到自己的 ID，另外，如果它要与集群中的其他节点进行通信，则从这个列表中发现。

`hostname` 是主机名或者当前操作系统上的 IP 地址，第一个 `port` 是用于集群节点间通信的端口，第二个 `port` 是在发起选举时使用的端口。如果该节点是 `observer`，则可以在最后追

加一个 *observer* 标识。如果节点不是 *observer*，则由选举产生 *Leader* 或 *Follower*，这两个角色的统称是 *participant*，即参与者。

#### 5) `peerType=observer`

如果该节点是一个 *observer* 节点，则需要添加此配置。

#### 6) `cnxTimeout`

在发起一次选举时集群中的各个节点首先需要建立一个连接，该选项设置了建立连接所需的超时时间，如果超时则重试。

## 8.2.4 动态重配置

ZooKeeper 的简捷架构帮助我们在分布式系统中很好地完成了分布式协调的任务，对于日常运维，系统管理员们不得不面临一个问题，那就是随着环境的变化对 ZooKeeper 生产配置的不断变更。

服务器硬件有异常时，需要用一个新设备替换集群中的故障节点；当性能出现瓶颈时，需要增加集群节点来提高吞吐量；当磁盘存储空间不足时，要将数据文件进行迁移；在实际的生产环境中，运维人员要根据类似于上面的情况对 ZooKeeper 配置文件进行变更，可能涉及 IP 地址、端口变化、存储目录变化等。而 ZooKeeper 在 3.5.0 版本之前配置文件中的设置是静态的，也就是说每次需要在编辑配置文件后重启服务以使之生效。对于运维人员来说在一个大集群中重新配置是非常痛苦的，除了手工编辑的工作繁冗，也会有集群节点的数量变化、容易引发数据不一致的情况出现，我们可以想象，在重启时，当一个数据没有完全同步的节点与新加入集群中的节点组成一个新的“大多数”，最终变成新 *Leader* 时，则意味着部分数据就此丢失了。另外，对于配置文件的变更，启动出错也有一定几率，如何回滚到变更前的状态就是运维人员要考虑的问题了。

对于 ZooKeeper 的配置变更要非常谨慎，运维人员手动地控制好集群节点的启动顺序并做好集群配置备份，每次的变更过程都是极其烦琐的。ZooKeeper 在 3.5.0 版本后，新增了动态重配置功能（Dynamic Reconfiguration），解决了 ZooKeeper 的静态配置修改的问题，同时将运维人员从重复且高风险的配置变更中解放出来。

### 1. `standaloneEnabled`

在 3.5.0 版本之前，ZooKeeper 的 *standalone* 和 *quorum* 运行模式的实现方式完全不同。在服务处于运行状态时，这两种模式无法进行切换，其结果是如果启动时我们采用的是 *standalone* 模式，则后续要动态地加入新节点并组成一个集群是无法实现的；另外，一个多节点的以集群 *quorum* 模式运行的 ZooKeeper，将它的节点数缩容到一个也是不允许的。如果有这种从集群多节点缩容到一个，或者从一个节点扩容到多节点的动态容量变化需求，

则需要修改以下配置：

在 ZooKeeper 的静态文件中，有如下配置项：

```
standaloneEnabled=false
```

其默认值为 `true`，将其修改为 `false` 后，只允许 ZooKeeper 运行在 `quorum` 模式下，这样在集群节点容量的配置上就有了更灵活的弹性。在使用 ZooKeeper 3.5.0 版本的动态重配置功能时，建议修改此参数的默认值。

## 2. 动态配置文件

ZooKeeper 从 3.5.0 版本开始，将配置参数分为静态与动态两种：静态参数在服务器启动时通过读取配置文件来获取，在运行时不能做任何变动；而动态参数允许在运行时被修改。这些动态参数主要有以下几个方面：服务节点、节点分组、节点权重。关于分组与权重在 8.2.5 节中会详细介绍。

ZooKeeper 将动态参数与静态参数存储在不同的文件中，通过在静态配置文件设置 `dynamicConfigFile` 参数链接到动态文件。下面是静态文件 `zoo.cfg`，其中引入了动态文件的链接：

```
tickTime=2000
dataDir=/app/zookeeper/data
initLimit=5
syncLimit=2
dynamicConfigFile=/app/zookeeper/conf/zoo.dynamic.cfg
```

动态文件 `zoo.dynamic.cfg` 的内容如下：

```
server.1=192.168.1.23:2780:2783:participant;2791
server.2=192.168.1.24:2781:2784:participant;2792
server.3=192.168.1.25:2782:2785:participant;2793
```

在这个示例文件中定义了三个服务器节点，与本节所描述的基本配置的变化在于：

`clientPort` 这个配置不再独立设置，而是用放在 `server` 行末尾的字段表示；另外，前文所述的节点角色分为 `observer` 和 `participant`。

通过这种方式启动集群，使用 ZooKeeper 命令行 CLI 连入一个节点，通过 `config` 命令查看动态配置参数：

```
[zk: 127.0.0.1:2791(CONNECTED) 3] config
server.1=192.168.1.23:2780:2783:participant;localhost:2791
server.2=192.168.1.24:2781:2784:participant;localhost:2792
server.3=192.168.1.25:2782:2785:participant;localhost:2793
version=600000003
```

集群的动态配置内容全部记录在了 `/zookeeper/config znode` 中，`config` 命令仅仅是对获取该 `znode` 内容的一个简单封装，我们可以像监听其他 `znode` 一样来监听 `/zookeeper/config` 内

容的变化。

### 3. 动态参数调整

ZooKeeper 动态参数的调整意味着集群从最初状态 S 迁移到 S'，为了保证最终数据的一致性，在 S→S' 状态迁移、动态重配置的过程中，需遵循以下原则。

- 在 S→S' 状态迁移前，动态配置 (/zookeeper/config) 的修改如同普通 znode 一样，需要被提交，即获得集群大多数节点的确认。
- 在 S→S' 状态迁移的过程中，冻结所有的事务提交，在这个过程中产生的事务都会被 Leader 记录下来，在后续迁移完成后再发起提交。
- 在 S→S' 状态迁移完成后，恢复服务，正常对外提供事务服务。

对配置动态修改有两种方式：增量 (Incremental) 与非增量 (Non-Incremental)。前一种在原配置上增加或删除节点，后一种通过全新的配置进行替换。

#### 1) incremental

增量方式的示例命令如下，在运行 ZooKeeper 客户端 CLI 工具后，使用 reconfig 命令进行配置：

```
> reconfig -remove 3 -add server.5=192.168.1.28:1234:1235:1236
```

以上示例是在原有集群的基础上删除 server.3，添加 server.5，其 IP 地址是 192.168.1.28，集群通信端口为 1234，选举端口为 1235，对客户端提供服务的端口为 1236。

对于要同时删除、添加多个节点的情况，可以通过逗号进行分割，示例如下：

```
> reconfig -remove 3,4 -add server.5=192.168.1.28:2111:2112:2113,6=192.168.1.29:2114:2115:observer;2116
```

#### 2) non-incremental

在非增量方式下，ZooKeeper 当前集群的动态配置项被新的配置完全覆盖，这些配置可以通过配置文件获取，也可以从命令行中输入：

```
> reconfig -file newconfig.cfg
> reconfig -members server.1=192.168.1.28:2780:2783:participant;2791,
server.2=192.168.1.29:2781:2784:participant;2792,server.3=192.168.1.30:2782:2785:participant;2793
```

第一行示例从 newconfig.cfg 配置文件中获取动态配置项，第二行示例直接从命令行中读取新的集群信息。

在 8.2.5 节中我们将介绍 ZooKeeper 集群的分组与权重配置，这两个参数都在动态配置项的行列。在使用增量方式动态重配置时无法有效地去修改分组、权重，而非增量方式由

于是从新的文件中读取的，因而受到支持。

## 8.2.5 监控

在前面的内容中我们已经熟悉了 ZooKeeper 的配置与运行工作，接下来要考虑的是对服务添加相应的监控。ZooKeeper 有两种主要监控方式：四字命令和 JMX。

### 1. 四字命令

四字命令是 ZooKeeper 提供的一组四字符命令，用来检查服务运行状态，它可以通过一些简单的工具来运行，例如 `telnet`、`nc` 等。四字命令返回的内容是可读的，这组命令集还在增长中，在这里介绍一些常用的命令集。

(1) `ruok`：这个命令是最简单的健康检查命令，服务运行正常时则返回 `imok`，否则将不做响应。这里的正常是相对的，例如服务节点可能正常地服务于客户端，但与其他集群节点的通信出现问题，在这种情况下该命令依旧返回 `imok`。为了获取服务节点的更多状态信息，可以使用 `stat` 命令。

(2) `stat`：`stat` 命令会输出更加详细的状态信息，包括当前服务的客户端连接数、服务角色及最新的事务 `zxid` 等。

(3) `svr`：除了提供 `connection` 信息，`svr` 命令提供的其他信息与 `stat` 相似。

(4) `dump`：提供所有 `session` 相关的信息，包括其超时时间及相关的暂时性 `znode` 节点，该命令只对 `Leader` 生效。

(5) `conf`：列出所有服务器启动时加载的基本配置信息。

(6) `envi`：列出 ZooKeeper 进程的所有环境变量。

(7) `mntr`：提供比 `stat` 更加详细的报表信息，每一行的内容以 `key-value` 键值对应的方式展示。

(8) `wchs`：列出服务器上相关的监听信息。

(9) `wchc`：列出服务器上相关的监听信息，通过会话进行排序。

(10) `wchp`：列出服务器上相关的监听信息，通过 `znode path` 进行排序。

下面使用 `nc` 工具来向服务器发送四字命令，进行监控：

```
$ echo mntr | nc localhost 2185
```

```
zk_version 3.4.0
zk_avg_latency 0
zk_max_latency 0
zk_min_latency 0
```

```

zk_packets_received 70
zk_packets_sent 69
zk_outstanding_requests 0
zk_server_state leader
zk_znode_count 4
zk_watch_count 0
zk_ephemerals_count 0
zk_approximate_data_size 27
zk_followers 4
zk_synced_followers 4
zk_pending_syncs 0
zk_open_file_descriptor_count 23
zk_max_file_descriptor_count 1024

```

使用 ruok 做简单的监控检查监控:

Here's an example of the ruok command:

```

$ echo ruok | nc 127.0.0.1 5111
imok

```

从 ZooKeeper 3.5.0 版本开始, 我们可以使用 HTTP 接口提交四字命令的请求。在 ZooKeeper 服务中嵌入了 Jetty server 作为 AdminServer 管理服务器, 其默认端口是 8080。服务访问的 URL 格式是 “/commands/[command name]”, 例如我们访问 stat 命令, 其命令接口是 <http://localhost:8080/commands/stat>, 服务器返回的结果是 JSON 格式。如图 8-3 所示。



图 8-3 HTTP 方式查看 ZooKeeper 状态

要查询当前版本支持哪些命令格式，请访问 <http://localhost:8080/commands>，如图 8-4 所示。

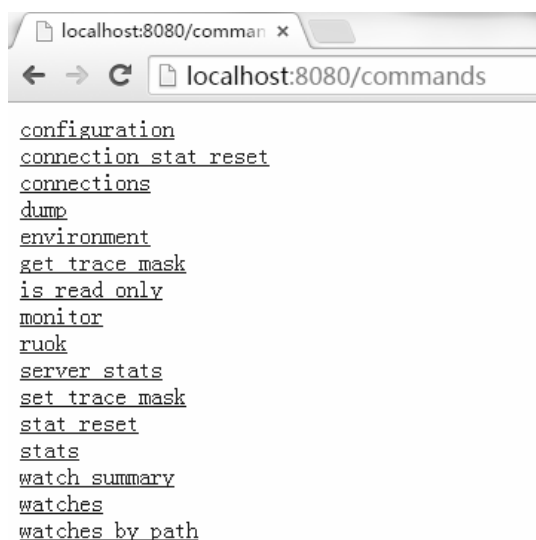


图 8-4 HTTP 方式查看 Zookeeper 命令

在静态配置文件中添加如下参数配置来管理服务器。

- (1) admin.enableServer: 设置为“false”为关闭管理服务器，默认为“true”。
- (2) admin.serverPort: 设置管理服务器端口，默认是 8080。
- (3) admin.commandURL: 设置命令查询的 URL 地址，默认是“/commands”。

## 2. JMX 监控

四字命令通过 TCP、HTTP 直接对外提供服务，由于这两个协议的通用性，以及大量的库文件支持二次开发，运维人员很容易将四字命令集成到监控平台中，但其也有薄弱的环节，那就是不支持远程 ZooKeeper 服务的管理与控制。

ZooKeeper 同时提供 JMX (Java Mangement Extensions) 标准的 Java 管理协议接口。关于 JMX 将在专门的监控章节中介绍，运维人员可以使用 JMX 库文件将 ZooKeeper 纳入客户化的监控平台。在这部分我们将使用 Java 工具库自带的 JConsole 通过 JMX 来访问 ZooKeeper。

JConsole 是随 Java 一起发布的管理工具，在 Java 根目录的 bin 下可以找到它。Java 1.6.20 之前的版本启动 JConsole 时需要提前设置环境变量 TEMP，否则启动会出现异常。如图 8-5 所示。

先启动 ZooKeeper 服务，之后单击 Jconsole 按钮，进入如图 8-6 所示的界面。





图 8-5 设置 TEMP 环境变量



图 8-6 JConsole 启动界面

在 JConsole 新建连接的界面上可以看到它已经发现了本地的 JVM 进程，其中包括 ZooKeeper。我们选择 PID 9784 进程，双击鼠标进入管理界面，如图 8-7 所示。

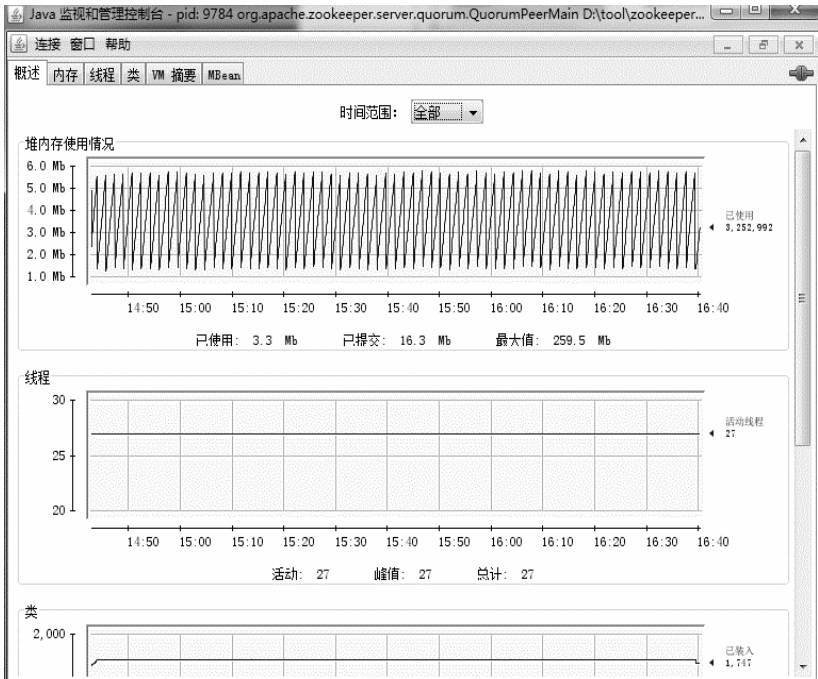


图 8-7 JConsole 监控界面

在界面中我们看到概述信息，这是 JAVA 提供的标准 MBeans 获取的数据，其中有内存使用情况、工作线程情况等。单击窗口的 MBean 一栏，所显示的内容如图 8-8 所示。

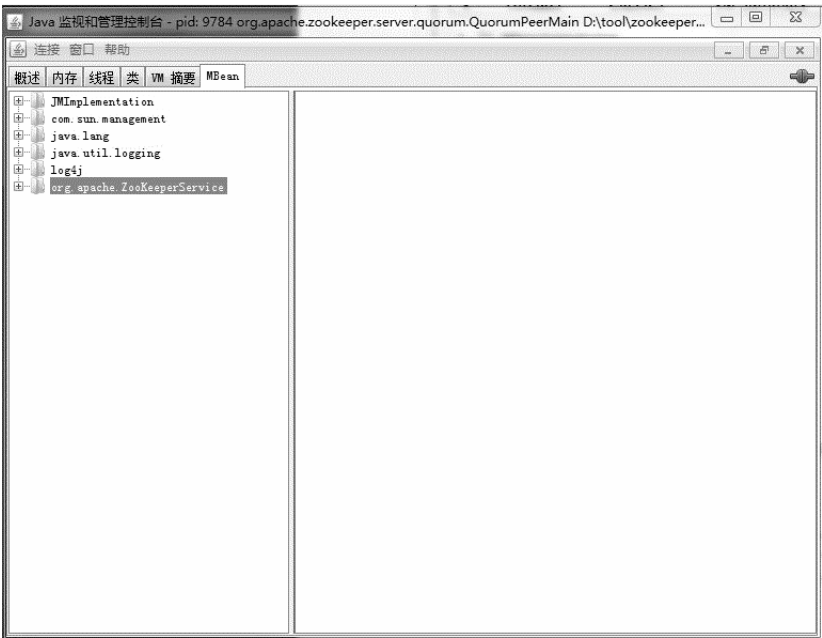


图 8-8 JConsole MBean 界面

该窗口列出了此 Java 向外提供的所有 MBean，值得注意的是 ZooKeeper 自定义的一行 org.apache.Zookeeper，双击该节点，进入下一层并找到 Connections 目录，可看到由一个本地客户端 127.0.0.1 连接到 ZooKeeper 的相关信息，如图 8-9 所示。

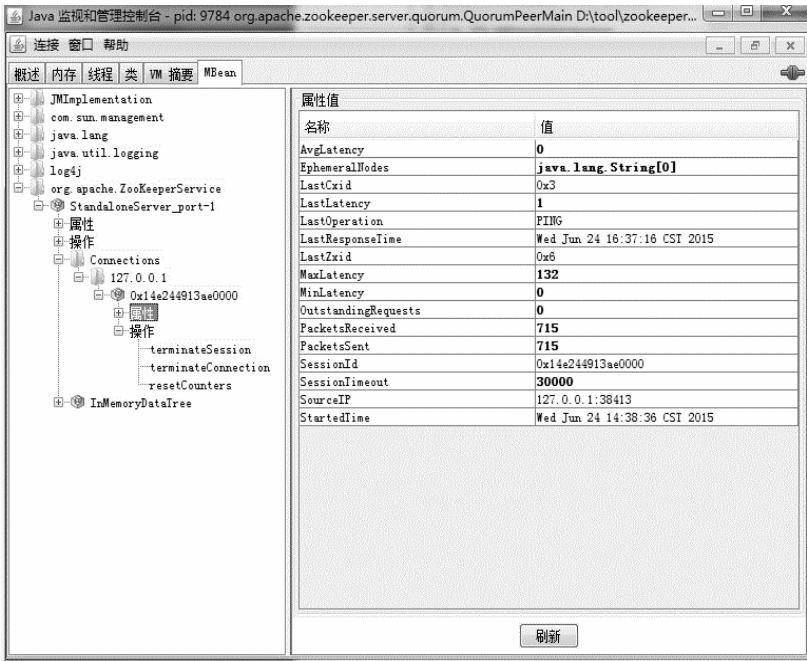


图 8-9 ZooKeeper 连接信息

JMX 与四字命令的不同是，它不仅提供对状态信息的查看，还能够进行相关操作与控制。单击 `terminateSession`、`terminateConnection` 后在窗口右侧有相应的按钮可以直接对 ZooKeeper 发起命令。其中 `terminateConnection` 会马上关闭服务器与客户端的网络连接，但依旧保持会话，客户端可以重新连接到服务器或者集群的其他节点上，其之前的状态信息及暂存节点都会保持。单击 `terminateSession` 按钮时，客户端与服务器的会话关闭，全部删除暂存节点。

JConsole 实例是与 ZooKeeper 服务在同一台机器上的本地访问，实际生产环境需要开启 ZooKeeper 服务的 JMX 远程访问。JMX 相关配置是在 Java 进程启动时以 Java option 启动参数设置的。可以在 ZooKeeper bin 目录 `zkServer.sh` 脚本中设置 `SERVER_JVMFLAGS` 环境变量，从而传递到 Java 启动参数中。下面是该环境变量的配置示例：

```
SERVER_JVMFLAGS=" -Dcom.sun.management.jmxremote.password.file=auth.
txt \
-Dcom.sun.management.jmxremote.port=6666 \
-Dcom.sun.management.jmxremote.ssl=false \
-Dcom.sun.management.jmxremote.access.file=access.txt "
```

在这个配置中涉及用户认证文件 `auth.txt` 及用户授权文件 `access.txt`。在进程工作目录中创建 `auth.txt` 文件，以下内容是示例说明：

```
yuhe zookeeperPaas
```

文件格式是一个键值对，分别表示用户名与用户密码，远程连接时需要输入用户名、密码。JMX 支持更高级的安全认证配置如 SSL 证书等。

同样，在用户授权上，在同目录下创建 `access.txt` 文件，输入以下内容：

```
yuhe readwrite
```

文件内容以键值对存在，在示例文件中用户 `yuhe` 具有读写权限。现在用户通过 JConsole 远程连接到 ZooKeeper 服务所在主机的 6666 端口，提供用户名、密码连接，并进行读写控制。

## 8.3 ZooKeeper进阶

在实际生产环境的 ZooKeeper 运行中，除了逻辑层上的部署与配置，还有其他诸多方面的考虑，例如物理服务器的放置。对于集群中的多节点，我们应当将其放置在不同的机架中；而对于跨 IDC 的节点放置，`tickTime` 时间要调整得更长些，以免网络的延时造成节点从集群中频繁移除、加入，从而影响 ZooKeeper 稳定。我们在本节先讨论 ZooKeeper 的分组与权重设置方法，再深入到分布式协调算法源头 Paxos，随后进入 ZooKeeper 内部，了解其自身的分布式算法 ZAB 协议。

### 8.3.1 分组与权重

在 ZooKeeper 的集群模式中，大多数集群节点在 Leader 选举及数据修改事务中发挥着重要作用。在 ZooKeeper 中使用了一个与现实生活中一致的词语替代——法定人数 (quorum)，在 ZooKeeper 中默认的法定人数是  $N/2+1$ ，法定人数要超过集群节点半数，这样才能够在同时产生两次选举、事务活动时至少有一个节点拥有最新数据、权威数据，让活动的结果保持一致，而不会出现脑裂。

在 ZooKeeper 的集群中有三类角色：Leader、Follower 与 Observer。Observer 与 Leader 同步最新的数据，向客户端提供读操作；Follower 除了同步数据、提供读操作，还要参与到“民主”生活中，在集群启动或者当前 Leader crash 时进行选举投票，在数据修改时进行同步提交；Leader 则以中央服务器模型接收转发过来的所有数据修改操作。

理想的民主是人人平等，人手一票，而现实社会的民主却并不是按照这种简单的逻辑处理的，某些法案、条例的投票及选举的参与者被分成了组，组中的成员投票有着不同的分量。

假设有一个现实而理想的民主国，它由多个州组成，各州由贵族与平民组成，每次领导大选先从州内投票开始，贵族与平民全部参与投票，由于贵族人数较少，但掌握更多的国家资源，为了保障公平，在权重上偏向于贵族，一个贵族的选票可以顶十个平民。州内投票发起后，采用“胜者全得”的方式，一旦获得大多数选票则全州统一选举对象。在州内完成选举后，记录所有州的选举对象，获得大多数选票者被称为领导。

ZooKeeper 中的选举在新版本中打破了之前的简单原则，没有再对“法定人数”进行约束，而是采用分组、权重的方式模拟现实的理想国度。回顾之前的选举过程，如果有  $G$  个分组，则必须有  $G' > G/2$  的选票才能当选。而在  $G$  个分组内部，有  $m$  个成员，每个成员有不同的权重，在所选权重  $w' > w/2$  时，才认为组内选举通过。

ZooKeeper 的分组配置如下，这些配置支持动态重配置：

```
group.x=n[:n]
```

group 后跟的 x 如同之前配置的静态文件 server 一样表示标识符号。等号右边的 n 代表在集群配置中的 server 标识符，所有的 server 都必须出现在各个 group 中出现，并且不能够重复。

下面是一个有 9 个节点的 ZooKeeper 集群，其被分为 3 组：

```
group.1=1:2:3
group.2=4:5:6
group.3=7:8:9
```

如果不设置权重，则所有节点权重默认相同，在这样一个层次化的集群中，需要两个组统一投票对象，而在组内需要两个成员统一投票对象才可以产生最终的选举结果。

设置权重的配置方式如下，其同样支持动态配置：

```
weight.x=n
```

weight 后面的 x 对应 server 的标识符，n 代表权重值，默认值为 1。

我们将 ZooKeeper 集群部署到不同的机架上，或者部署到不同的数据中心时，可以根据环境情况对其进行分组，设置不同的权重。

### 8.3.2 Paxos 算法

在分布式系统环境中，节点之间通过消息传递进行通信，基于消息的分布式系统引入了多方组件，发送节点、接收节点、路由节点等，传输网络的任意节点在传输过程中都可能宕机、处理缓慢从而导致消息延迟、丢失甚至重复。Paxos 算法要解决的问题就是在假设以上异常存在的情况下，集群中所有的计算节点对某个全局数据的值达成一致，并且要满足数据的一致性。ZooKeeper 在 Paxos 的基础上开发了自己的 ZAB 协议，该部分让我们回溯源头，了解与掌握 Paxos 算法。

Paxos 是莱斯利·兰伯特（Leslie Lamport）在 1990 年提出的一种保证消息传递型系统中数据一致性的算法。为了描述 Paxos 算法，Lamport 虚拟了一个叫作 Paxos 的希腊城邦，这个岛按照议会民主制的政治模式制订法律，由于所有参与民主制的人都有自身的职业与工作，因此没有人愿意将自己的全部时间和精力放在这种事情上。无论是议员、议长还是传递纸条的服务员都不能承诺在别人需要时自己一定会出现，也无法承诺批准决议或者传递消息的时间。议员、议长们只要等待足够的时间，消息就会被传到。另外，Paxos 岛上的议员是不会反对其他议员提出的决议的。

Lamport 戏剧性地将分布式系统中的工作环境映射到了一个虚拟的现实世界中。对应于分布式系统，各个节点对应于议员，系统的状态对应于制订的法律。节点和消息传递通道的不可靠性对应于议员和服务员的不确定性。分布式系统与这个虚拟的城邦一样，需要对一个状态或者一个法律达成一致，一个法律条文只有一个版本，而在同一时间读到条文的人看到的内容是一致的。

关于 Paxos 论文的发表还有一段有趣的故事。在 1990 年，Lamport 用这种故事阐述的方式将 Paxos 算法提交给了 ACM TOCS，当时的三名评审委员并没有完全理解 Paxos 算法的重要性，他们给 Lamport 的反馈是“这篇论文非常有趣，虽然它并不重要”，同时他们要求 Lamport 删除 Paxos 城邦的故事部分，采用算术公式来表示。Lamport 是一个非常富于人文气息的理论家，他对委员会的反馈非常生气，并拒绝了论文的修改。在多年之后，SRC 的一些人正在寻找一个算法来支持他们的分布式系统，Lamport 将论文给了他们，在实践中证明 Paxos 的重要性，并就论文进行数学符号的修饰，直到 1998 年的 ACM TOCS 上，Paxos 算法才重新被接受，这时距论文发表已过了 9 年。

## 1. 算法

Paxos 算法解决的问题是如何在分布式环境中保证数据的一致性。它将所有节点分为 proposer、acceptor 和 learner 三种角色。proposer 提出提案，提案信息包括提案编号和提议的值 value，被称为决议；acceptor 收到提案后可以接受（Accept）提案，若提案获得多数 acceptors 的接受，则称该提案被批准（Chosen）；learner 只能“学习”被批准的提案。划分角色后，我们将问题重新定义。

（1）决议（Value）只有在被 proposer 提出后才能被批准（未经批准的决议被称为“提案”，即 proposal），批准之后的决议被称为批准。

（2）在一次 Paxos 算法的执行实例中，只批准一个 value。

（3）learner 只能获得被批准的 value，即获得达成一致的数据。

一个决议的通过可分为两个阶段。

### 1) prepare 阶段

（1）proposer 选择一个提案编号  $n$  并将 prepare 请求发送给 acceptor 中的一个多数派。这里的多数派指超过了 acceptor 总数的一半，这样可以保证不同的 proposer 提出同一个提案的不同决议时，至少有一个公共的 acceptor 会批准唯一的一个决议；

（2）acceptor 收到 prepare 消息后，如果之前没有接受过其他提案，则直接回复 OK，如果接受过其他提案，但新提案编号大于旧的，则 acceptor 将自己上次接受的提案决议及提案编号回复给 proposer，并承诺不再回复小于  $n$  的提案；如果接受过其他提案，并且提案编号小于旧提案，则 acceptor 不做任何回复。

### 2) 批准阶段

（1）当一个 proposer 收到了多数 acceptor 对 prepare 的回复后，就进入批准阶段。它要向回复 prepare 请求的 acceptor 发送 accept 请求，要求包括编号  $n$  和 value 的提案。这里值得注意的是 proposer 发出的决议不一定就是自己提案中的决议，因为从 prepare 中收到 acceptor 的回复中，包含了它们批准了的决议，这时，proposer 会遵循规则，从这些带有决议中选择一个编号最大的发送给 acceptor，但编号保持不变。

（2）在不违背自己向其他 proposer 承诺的前提下，acceptor 收到 accept 请求后即接受这个请求。

将上面完整的算法映射到 Paxos 图解，如图 8-10 所示。

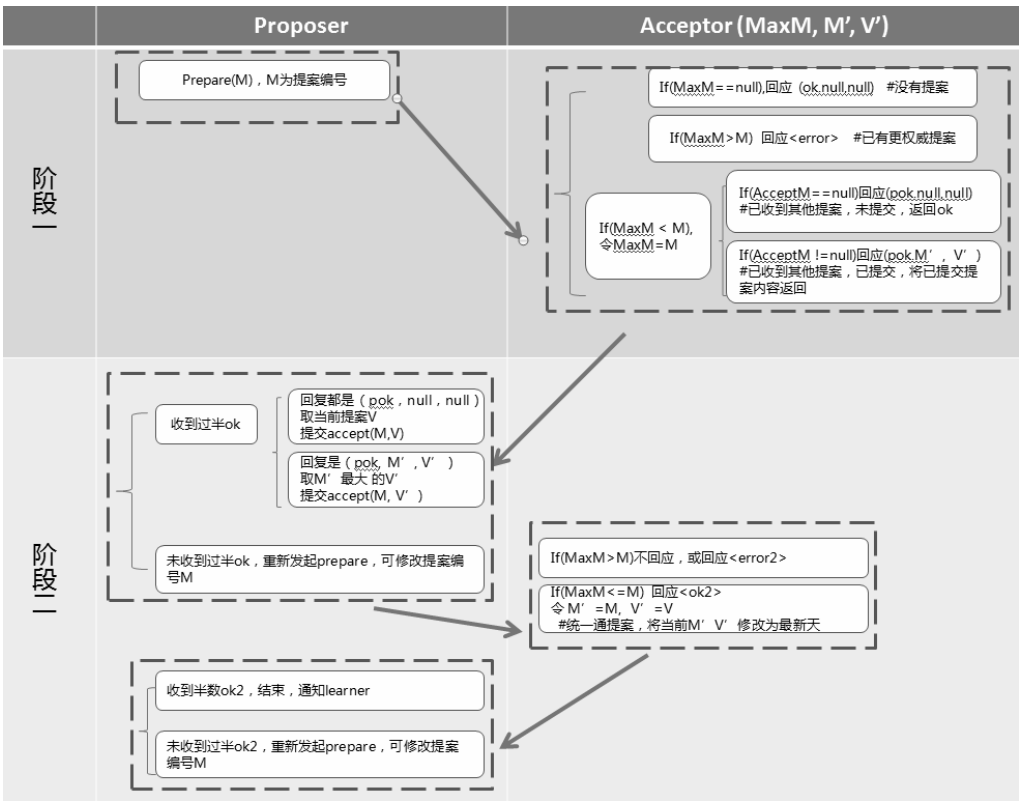


图 8-10 Paxos 算法图解

在整个过程中最重要的两个角色是 `proposer`、`acceptor`，每次的决议过程有两个阶段，每个阶段有两个步骤，阶段一有 `prepare`、`promise` 步骤，阶段二有 `accept`、`accepted` 步骤。决议（`value`）在被接受之前都是以提案状态存在的，这是一个状态，一旦被接受，则变为批准。在一个提案中包括了编号与决议。我们看到，一些步骤会依据编号的大小来判断是否回复 `proposer`，编号对最后一致性的数据结果有着重要影响，编号的生成规则并不在算法定义之中。你可以类比现实世界，某些 `proposer` 的地位比其他高，在这个提案还没有被通过时，他具有更多的发言权。

2. 示例

我们通过一个经典的例子来描述上述过程：有 `A1`、`A2`、`A3`、`A4`、`A5` 5 位代表，他们就纳税税率的问题进行决议，议员 `A1` 决定将税率定为 10%，因此他向所有人发出一个草案，这个草案的内容是：

现有的税率是什么？如果没有决定，则建议将其定为 10%。时间：本届议会进行于第 3 年 3 月 15 日；提案者：`A1`。

在最简单的情况下，没有人与其竞争；信息能及时、顺利地传达到其他议员处，于是 `A2-A5` 回应：

我已收到你的提案，等待最终批准。

而 A1 在收到两份回复后就发布最终决议：

税率已定为 10%，新的提案不得再讨论本问题。

现在我们假设在 A1 提出提案的同时，A5 决定将税率定为 20%：

现有的税率是什么？如果没有决定，则建议将其定为 20%。时间：本届议会第 3 年 3 月 15 日；提案者：A5。

我们假设 A1 的提案送给了 A1、A2、A3，而 A5 的提案提交给了 A3、A4、A5，根据算法，A1、A2、A4、A5 将回复他们接受的提案，而 A3 由于收到了两份提案，他如何回复以及最终会批准哪一个，这里提案的编号就发挥很大的作用了，我们分三种情况考虑。

### 1) 情况一，A5 在批准之后送达提案

假设 A1 的提案先送到 A3 处，而 A5 的侍从决定放假一段时间，于是 A3 接受并派出了侍从，A1 等到了两位侍从，加上它自己已经构成一个多数派，于是税率 10% 将成为决议。A1 派出侍从将决议送到所有议员处：

“税率已定为 10%，新的提案不得再讨论本问题。”

A3 在很久以后收到了来自 A5 的提案，由于税率问题已经讨论完毕。他决定不再理会，但是他要抱怨一句：

“税率已在之前的投票中定为 10%。你不要再来烦我！”

这个回复对 A5 可能有帮助，因为 A5 可能因为某种原因很久无法与外界联系了，当然更可能对 A5 没有任何作用，因为 A5 可能已经从 A1 处获得了刚才的决议。

### 2) 情况二，A5 在批准前送达提案

依然假设 A1 的提案先送到 A3 处，但是这次 A5 的侍从不是放假了，只是中途耽搁了一会儿，这次，A3 依然会将“接受”回复给 A1，但是在决议成形之前它又收到了 A5 的提案。

如果从提案编号上辨别出 A5 是一个权威人物，则 A3 不敢怠慢，忽略之前 A1 提交过但未批准的提案，立即回复 A5：

“我已收到你的提案，等待最终批准。”

A1 在第二阶段要求 A3 通过提案时，A3 因为接受了 A5 的提案，所以对 A1 不做回复，或回复更权威的提案。A5 在收到 A3 的回复后进入下一阶段，最后的提案通过后就变成决议。



### 8.3.3 ZAB协议

ZooKeeper 的一致性能保证算法使用的是 Paxos 吗？答案是否定的，学院派与实践派之间存在着天然的差别，学院派的理论应尽量适应所有的场景，囊括一个更大的问题域，但正是因为这种通用性要求而使设计变得复杂。而实践派则会选择一个特定场景，依据具体的需求而精简算法，让整体架构更加简单。

ZooKeeper 在架构设计上对所有数据的修改只保留一个节点，我们称之为 Master 或者 Leader，所有的提案都由 Leader 发起，其充当了 Paxos 第一阶段中的 Proposer、Acceptor。一旦有新的提案，Leader 会直接通过广播形式传递给集群中的 Follower，只有大多数 Follower 同意了提案，Leader 才将提案通过并提交。ZooKeeper 的集中式 Leader 相当于“独裁”式架构，所有的数据变动都必须经过 Leader，而不是在多个同级节点之间交互，这样让问题变得更加简单、高效。但在容错上考虑，一旦 Leader 瘫痪，整个集群将停止工作，为了弥补这种缺陷，ZooKeeper 采用了一种简单的选举动作以在 Leader 瘫痪后迅速地从 Follower 中选出新的 Leader。再回到性能上考虑，如果并发量大，所有的数据变动都要经过 Leader 并同步，则这样在高并发的场景下 ZooKeeper 是否适用呢，ZooKeeper 明确的答复是其使用的场景聚焦在分布式协调问题上，而不是分布式高速缓存、消息队列，ZooKeeper 也能够快速地扩充集群节点，但 Follower 是作为读节点存在的。

ZooKeeper 使用了一种被称为 ZAB (Zookeeper Atomic Broadcast) 的协议作为其一致性数据保证的核心，该协议的运行过程也有两个阶段：选举阶段、广播阶段，两个阶段随着某些情况的发生而不断变化。

#### 1. 第一阶段——选举阶段

当一个 ZooKeeper 中的一个节点启动时，其最初状态为 LOOKING，它必须寻找一个 Leader。如果在集群中存在 Leader，则它通过询问其他 server 节点获知 Leader 信息，并与 Leader 建立连接，随后进行数据同步。

但它发现集群中所有的节点都是 LOOKING 状态时，这些节点必须发起一次选举活动，互相交换信息，从中产生一个 Leader，获得最多选票的 Leader 进入 LEADING 状态，而其他节点进入 FOLLOWING 状态。

选举动作类似于两阶段提交，在集群中通过两次信息交换而产生出 Leader。第一次集群中的节点将自身信息发送到集群中，这个信息包括 server.id 和 zxid。server.id 是节点的服务标识符，zxid 是 ZooKeeper 事务 ID。

ZooKeeper 状态的每一次改变，都对应着一个递增的 Transaction id，该 ID 叫作 zxid。由于 zxid 的递增性质，如果 zxid1 小于 zxid2，那么 zxid1 肯定先于 zxid2 发生。创建任意节点，或者更新任意节点的数据，或者删除任意节点，都会导致 ZooKeeper 的状态发生改变，从而导致 zxid 的值增加。

zxid 是一个 64bit 的整型 ID，它由两部分组成：epoch 和 counter，每部分都有 32bit 长，counter 是递增的 Transaction id，是后半部分；而 epoch 作为前半部分，代表着集群中 Leader 所改变的次数。在一个集群运行的全过程中，发生的 Leader 选举切换都会在这个 epoch 中体现，每改变一次递增 1。

在 ZooKeeper 的 Leader 选举时，其处理过程如下。

(1) 将自己的 server.id 和 zxid 信息发送出去。

(2) 如果收集的信息中  $\text{receive.zxid} > \text{my.zxid}$ ，或者  $\text{receive.zxid} = \text{my.zxid}$ ，但  $\text{receive.server.id} > \text{my.server.id}$ ，则将自己的选票修改为  $(\text{receive.zxid}, \text{receive.server.id})$ ，否则是  $(\text{my.zxid}, \text{my.server.id})$ 。

(3) 一旦收到的选票超过半数的服务器节点，便将状态修改为 LEADING，并通知投票者将状态改为 FOLLOWING。

### 2. 第二阶段——广播阶段

同一时刻存在一个 Leader 节点，其他节点被称为“Follower”，如果是更新请求，客户端连接到 Leader 节点，则由 Leader 节点执行其请求；如果连接到 Follower 节点，则需将转发请求到 Leader 节点执行。但对读请求，客户端可以直接从 Follower 上读取数据，如果需要读到最新数据，在发起读操作时设置强制同步，则会从 Leader 节点获取。

考虑到 ZooKeeper 主要操作数据的状态，为了保证状态的一致性，ZooKeeper 提出了以下两个安全属性。

- 全序 (Total Order)：如果消息 a 在消息 b 之前发送，则所有 Server 应该看到相同的结果。
- 因果顺序 (Causal Order)：如果消息 a 在消息 b 之前发生 (a 导致了 b)，并被一起发送，则 a 始终在 b 之前被执行。

为了保证上述两个安全属性，ZooKeeper 使用了 TCP 和 Leader。使用 TCP，以字节流的方式发送数据，在 TCP 内部进行数据排序，既保证了全序，也具备了可靠性。而通过 Leader 负责所有的写操作，则解决了因果顺序问题，先到 Leader 的先执行。因为有了 Leader，ZooKeeper 的架构就变为 Master-Slave 模式，而集群 Server 间的模式是 Peer-to-Peer。ZooKeeper 采用了一种“折中”的架构方式让问题变得简单。

广播阶段的 ZooKeeper 数据同步与 Paxos 的两个阶段相似，但 ZooKeeper 简化了很多，由于所有的数据提交修改入口都是 Leader，所以它的 Proposer 就只有一个，而每一次数据修改动作都是完整的原子操作，在一个 znode 操作还没有写完之前，下一个修改是被 Leader 这个入口所控制的，因此 Leader 提交到所有 Follower 节点的数据请求并不用担心有其他竞争者。

### 8.3.4 分布式协调场景

#### 1. 全局配置

ZooKeeper 提供的一个最直接功能即全局配置，分布式环境中的节点可以将全局性的配置信息写入 ZooKeeper 集群中，并通过共享被其他节点读取。正因为 ZooKeeper 的这个最基本的功能，在它之上才可以由应用自行组合并适应更复杂的使用场景。

#### 2. 锁

锁是在操作系统中引入的对全局资源的保护机制，对于一个临界区同时只能有一个任务进入，其他任务只有等前一个任务退出后才能进入。锁的获取与释放动作本身要保证原子性，对于一个请求锁的用户，它只能看到两种状态：空闲与锁定。ZooKeeper 的中央管理方式让锁的实现更加简单，但在 ZooKeeper 的集群中，数据副本在多个节点中都存在，这时对“锁”的数据强一致性非常重要。

#### 3. 两阶段提交

在分布式系统中，事务往往包含多个参与者的活动，单个参与者的活动是能够保证原子性的，而多个参与者之间原子性的保证则需要通过两阶段提交来实现，两阶段提交是分布式事务实现的关键。两阶段提交保证了分布式事务的原子性，这些子事务要么都做，要么都不做。

两阶段提交的过程涉及协调者和参与者。协调者可以看成事务的发起者，同时是事务的一个参与者。对于一个分布式事务来说，一个事务是涉及多个参与者的。具体的两阶段提交过程如下。

##### 1) 第一阶段

首先，协调者在自身节点的日志中写入一条日志记录，然后向所有参与者发送消息 `prepare T`，询问这些参与者（包括自身）是否能够提交这个事务。

参与者在接收到这个 `prepare T` 消息后，会根据自身的情况进行事务的预处理，如果参与者能够提交该事务，则会将日志写入磁盘，并返回给协调者一个 `ready T` 信息，同时自身进入预提交状态；如果不能提交该事务，则记录日志，并返回一个 `not commit T` 信息给协调者，同时撤销在自身上所做的数据库修改。

参与者能够推迟发送响应的时间，但最终还是需要发送的。

##### 2) 第二阶段

协调者会收集所有参与者的意见，如果收到参与者发来的 `not commit T` 信息，则标志着该事务不能提交，协调者会将 `Abort T` 记录到日志中，并向所有参与者发送一个 `Abort T` 信息，让所有参与者撤销自己的所有预操作。

如果协调者收到所有参与者发来的 `prepare T` 信息，则协调者会将 `commit T` 日志写入磁盘，并向所有参与者发送一个 `commit T` 信息，提交该事务。若协调者迟迟未收到某个参与者发来的信息，则认为该参与者发送了一个 `VOTE_ABORT` 信息，从而取消该事务的执行。

参与者接收到协调者发来的 `Abort T` 信息以后，参与者会终止提交，并将 `abort T` 记录到日志中；如果参与者收到的是 `commit T` 信息，则会对事务进行提交，并写入记录。

在一般情况下，两阶段提交机制都能较好地运行，当在事务进行过程中有参与者宕机时，该参与者重启以后，可以通过询问其他参与者或者协调者，从而知道这个事务到底提交了没有。当然，这一切的前提是各个参与者在进行每一步操作时，都会事先写入日志。

两阶段提交最大化地保证了分布式事务的成功性，它将各个事务参与者的“确认”时间缩至最小，我们也可以看到这其中是有可能存在问题的。若协调者在发出 `commit T` 消息后宕机了，而唯一收到这条命令的一个参与者也宕机了，这时这个事务就处于一个未知状态，没有人知道这个事务到底提交了没有，从而需要数据库管理员的介入，防止数据库进入一个不一致的状态。当然，如果有一个前提是所有节点或者网络的异常最终都恢复了，那么这个问题就不存在了，协调者和参与者最终会重启，其他节点最终也会收到 `commit T` 的信息。

### 4. 组成员

组成员能够自动加入、退出集群，组成员信息能够动态地分发给订阅者，在分布式场景下将任务分派给组成员。

### 5. Master 选举

为了保证应用的高可用性，有一种部署结构是扁平等价的多个节点，其中只有一个 **Master** 对外提供服务，需要有一种机制从所有等价节点中选举出一个，并且在这个 **Master** 出现异常后，剩余的节点能够发起一次选举，推举出新的 **Master**。

# 资源管理 Mesos

## 9.1 Mesos介绍

数据中心的资源管理是分布式 PaaS 平台的核心所在，从 20 世纪 60 年代 IBM CP-40 的出现到 20 世纪 90 年代后期 VMware 的崛起，虚拟机技术在数据中心资源管理历史中发展了相当长的时期，随着 VM 的广泛传播，于 2000 年兴起的云计算基本上是构建在虚拟化上的，从中都可以看到虚拟机的影子。但随着容器的兴起，以 VM 作为资源管理计算单元的局面被打破，谷歌 PaaS 平台 Kubernetes 对外开源时，其声称“谷歌内部一切都是容器”，而这些容器资源是如何将计算、存储、网络资源汇聚在一起管理的呢？

### 9.1.1 资源管理需求

在传统 IT 到互联网的转变过程中，数据中心的资源情况发生了很大的变化。第一个变化是软件的种类越来越多。传统 IT 所使用的软件种类相对较少，主要是 ERP 应用系统应用服务器、核心数据库等。在 IT 运营管理的成熟过程中，软件会逐渐标准化、单一化，慢慢形成主流组件，运维管理集中在控制好这批标准组件的稳定性、安全性。随着互联网的转型，海量用户访问、高并发量请求、大数据分析的需求接踵而至，在原有标准软件类型的基础上涌现了批处理、流式计算、高性能分析等各种类型计算，引入的中间软件也变得非常多。我们可以看到 Hadoop、Spark、Storm 等数据处理、分析型软件的兴起，而在 Service 计算类型上标准也逐渐被打破，Python、Ruby、Java 等各类开发语言实现的应用部署到了数据中心的各个位置，其中还要加入类似于代理、缓存等功能组件。除了软件种类的变化，高并发量、海量数据也直接反映在对基础资源剧增的需求上，IDC 必须提前准备好大量基础资源，应对随时而来的访问请求。

在传统模式下资源管理方式是将物理的或虚拟的操作系统分配给用户，使用配置管理工具来安装、部署软件，通过标准化、自动化方式简化处理流程。而在互联网下，开发人员对于软件需求不再拘泥于固定准则，软件的标准已被打破，Docker 的计算单元打包可解决软件标准化的问题。而面对大量的基础资源需求，IDC 的传统资源分配方法将不再适应，

用户无法等待虚拟机的创建，也无法接受手工复制计算单元到操作系统上的低效。应用类型有批处理、大数据、用户服务等，这些计算类型在静态的操作系统上进行了隔离，彼此之间的资源无法共享，支撑业务所使用的基础资源越多，对 IDC 的资源浪费越大。

资源管理器首要解决的问题是提高资源利用率。

在海量基础资源的需求下，如何提高资源利用率将直接影响到 IT 运营成本。在传统模式下资源管理方式是将物理的或虚拟的操作系统分配给用户，使用配置管理工具来安装、部署软件，通过标准化、自动化方式简化处理流程，这种静态资源的分配方式的最大问题是无法最大化地实现资源共享。如图 9-1 所示，在数据中心中有三个大的集群：用于大数据计算的 Spark，用于集成测试的 Jenkins，用于生产服务的 Tomcat。每个集群的服务时间跨度并不是 24 小时的，由于静态隔离，资源无法得到有效利用，当 Tomcat 集群中有节点发生异常时，其他集群即便有闲置资源也无法用来给 Tomcat 集群使用。

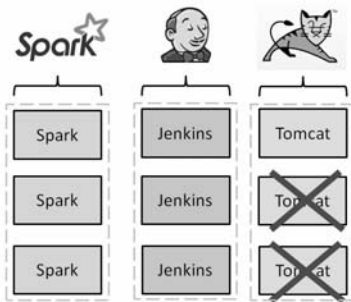


图 9-1 数据中心资源得不到利用

如图 9-2 所示，Mesos、Borg 等资源管理器实现了数据中心资源在各类应用中共享。资源管理器提升资源利用率的方法是打破这种静态隔离，打破基础资源在操作系统层的隔离，多种计算类型混合部署。在资源高度共享的同时要具备资源隔离性，即便是在同一台服务器上，不同租户的应用也不能互相影响。整个数据中心最终抽象成一个分布式操作系统内核，自动统一分配 IDC 中的相关资源，而不管具体的申请用途，共享让使用率更高，隔离确保应用安全、自动分配、提升伸缩性。

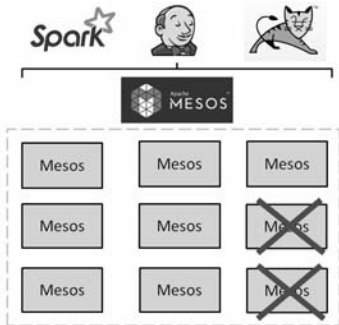


图 9-2 通过资源管理器 Mesos 实现资源共享

## 9.1.2 Mesos的起源

Mesos 是 Apache 下的开源分布式资源管理框架，它被称为分布式系统的内核。最初是由加州大学伯克利分校的 AMPLab 开发的，后来在 Twitter 得到广泛使用。数据中心资源管理器的鼻祖起源于 Google 的 Borg，它如同内部的中央大脑一样控制着任务在整个数据中心节点上运行，不同于为每一个软件服务建立一个独立集群（一个为搜索，一个为 Gmail，一个为地图），Borg 能够在集群上同时运行不同类型的工作，所有这些工作被细分为小的任务，Borg 将这些任务发送到它能够找到的空闲资源上，例如处理能力、计算机内存及存储空间。在 Mesos 项目启动一年后，项目创始人 Hindman 带着他的伯克利同学到 Twitter 进行了一次沟通，当时到场的 Twitter 员工只有 8 个人，Hindman 相当失望，但随后这 8 个人中的 3 个人加入了项目，他们都是之前的谷歌员工。他们告诉 Hindman，之前他们错过了 Borg，现在 Mesos 似乎是在用一种更完美的方式重建。

## 9.2 Mesos架构与 workflow

### 9.2.1 Mesos架构组件

Mesos 架构组件有三个重要组成部分：Master、Slave 及 Framework。Master 与 Slave 并不是互为主备的关系，Master 是总控节点，Slave 是计算节点，二者承担着完全不同的职责。

#### 1. Master

Master 负责将资源分配到不同的 Framework 中，并管理着任务的生命周期，它扮演着资源代理的角色，采用细粒度的资源分配方式，资源分配策略可以通过插件的方式进行变换。

#### 2. Slave

Slave 的职责是执行 Framework 发送过来的任务，它是真正的基础计算资源。Mesos 用资源与属性来描述所管理的 Slave 节点。资源是 Slave 能够被任务所消费的元素，而属性则是对 Slave 进行标注。这些属性内容可以是 Slave 节点的配置信息，包括物理位置、OS 版本、软件情况等。在 Mesos 集群中，Master 将基础资源分配给所有的 Framework，利用 Slave 的属性配置，我们可以做到指定的资源分配规则。在 Slave 启动时可以指定相关的选项：--resources 和--attributes，例如：

```
--resources='cpus:30;mem:122880;disk:921600;ports:[21000-29000]; '  
--attributes='rack:rack-2;datacenter:europa;os:ubuntu14.4'
```

以上表示 Slave 提供 30 Core CPU、120G 内存、900GB 磁盘、端口范围为 slave21000～29000。在属性信息上，其标注了物理位置在 rack-2，数据中心是 Europe，OS 版本是 Ubuntu 14.4。

### 3. Framework

严格地说，Framework 并不是 Mesos 的一部分，Mesos 与 Hadoop 1.0 在资源管理上的最大不同是它是一个二层资源管理架构，它将资源管理与任务调度的职能分开，这样可保证框架的足够灵活。Framework 的主要职责为调度不同类型的任务。

Mesos 提供的是 Framework 的接口，而不是一个具体的 Framework 实现。Mesos 为开发人员提供了一套用于构建大型可伸缩的分布式系统的 SDK，其中包括资源获取、部署、监控与隔离等。

运行在 Mesos 之上的分布式应用都可以被称为 Framework。Framework 由两部分组成，一个是调度器，一个是执行器。调度器负责与 Mesos Master 进行通信获取资源，并协调 Slave 上的执行器工作，执行器负责具体的任务执行。

Mesos 的整体架构如图 9-3 所示。

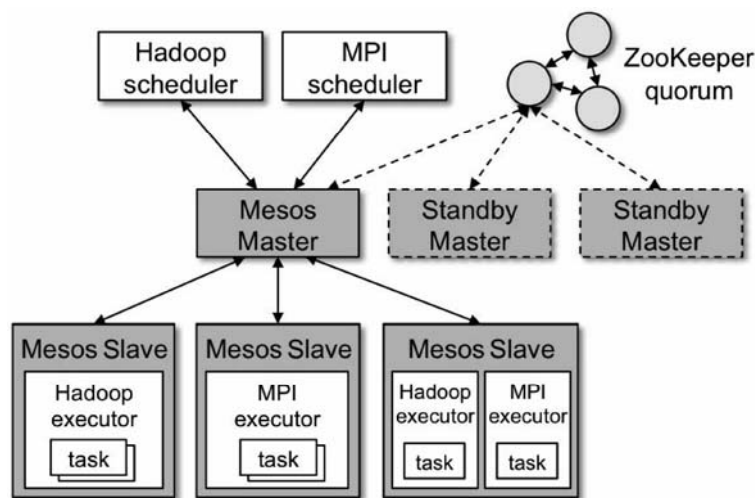


图 9-3 Mesos 的整体架构

在架构中央的是 Master，它们是数据中心的大脑，在一个高可用的架构中可以部署多个 Master 节点，一主多备，使用 ZooKeeper 分布式协调系统在其启动、异常时选举主节点。数据中心的所有基础资源以 Slave 节点的方式加入到集群中。资源管理与任务调度是分离的，任务调度由 Framework 负责，不同的 Framework 为不同计算类型服务，可以是 Hadoop、Spark，也可以是一般的 Web 服务。

## 9.2.2 Mesos资源管理的工作流程

Mesos 资源管理的工作流程如图 9-4 所示。



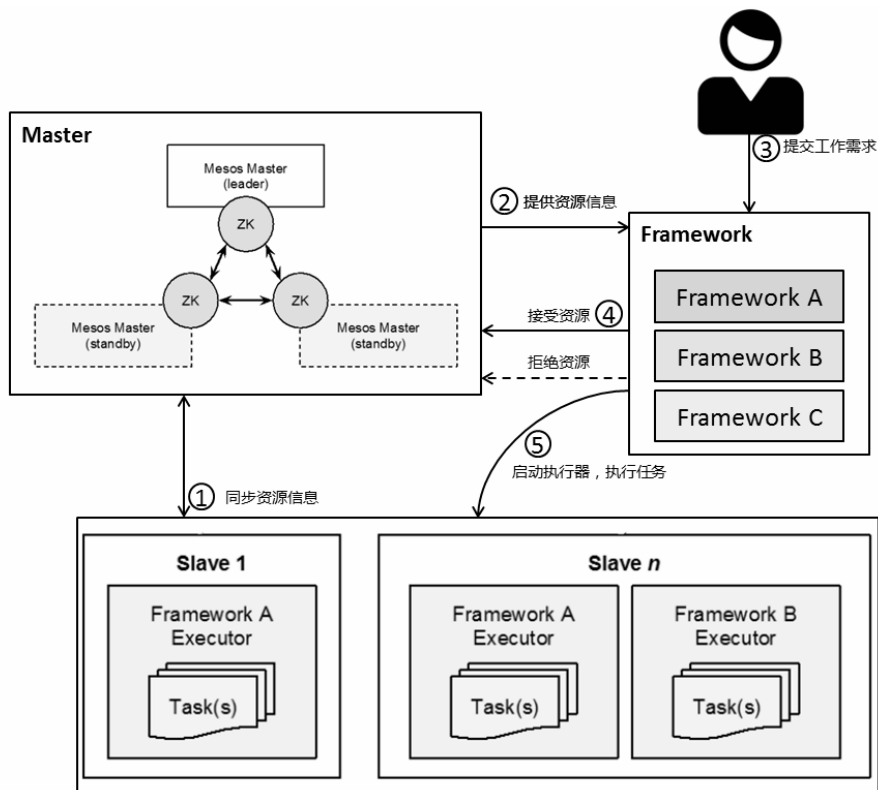


图 9-4 Mesos 资源管理的工作流程

Mesos 体系结构分为三大部分：Master、Slave、Framework。

(1) 集群中的所有 Slave 节点会与 Master 定期进行通信，将自己的资源信息同步到 Master。Master 由此获知整个集群中的资源状况。

(2) Master 会与已注册、受信任的 Framework 进行交互，定期将最新的资源情况发送给 Framework。当 Framework 前端有工作需求时，将选择接收资源，否则拒绝。

(3) 前端用户提交了一个工作需求给 Framework。

(4) Framework 接收 slaveMaster 发过来的资源信息。

(5) Framework 依据资源信息向 Slave 发起任务启动命令，开始调度工作。

关于 Mesos 分配算法的内容在后面的章节中会介绍到。集群中的资源收回、分配是一个持续的过程，Mesos 提供的资源 API 让 Framework 可以独立地完成资源扩容、伸缩。Framework 不仅可以接收资源，还可以拒绝接收。这一行为可以让 Framework 等待满足其要求的资源到位，比如特定的硬件服务器等。

Mesos 是典型的二层架构，Master 只负责资源管理，Framework 负责任务调度，它们二者都可以直接与 Slave 通信。Master 保持着通用性、轻量性，它与 Framework、Slave 之间的

交互更多的是状态信息的同步。在 Slave 上有两类进程：一类是 Manager，用来与 Master 通信，交互状态信息，另一类则是由 Framework 实现的具体 Executor，它负责任务的运行。对于 Manager 而言，它看到的所有 Executor、Task 都是一致的容器，而不管这些任务具体执行什么业务逻辑。Manager 进程异常仅代表其与 Master 之间的状态信息错误，并不会直接影响到其上的任务进程。

## 9.3 Mesos 安装配置

### 9.3.1 安装预先准备

在安装 Mesos 之前，需要提前安装以下程序包：

- g++(>=4.1)
- Python 2.6 developer packages
- Java Development Kit (>=1.6) and Maven
- The CURL library
- The SVN development library
- Apache Portable Runtime Library (APRL)
- Simple Authentication and Security Layer (SASL) library

如果是从 slavegit 仓库开始构建的，则还需要 autoconf (version1.12)、libtool。在不同的操作系统上有不同的安装步骤，在本书中我们将在 CentOS 6.5 和 Ubuntu 14.10 上进行安装。

#### 1. CentOS

我们使用下面的命令在 CentOS 上安装以上依赖包。

在 CentOS 默认仓库中并不提供 SVN 库 (>=1.8.)，因此在安装时需要添加一个仓库地址，我们在/etc/yum.repos.d中添加一个 wandisco-svn.repo 文件，在其中加入以下内容：

```
[WandiscoSVN]
name=Wandisco SVN Repo
baseurl=http://opensource.wandisco.com/centos/6/svn-1.8/RPMS/$basearch/
enabled=1
gpgcheck=0
```

现在我们可以使用以下命令来安装 libsvn：

```
centos@master:~ $ sudo yum groupinstall -y "Development Tools"
```

下面的命令下载、解压、安装 Maven，我们将其解压到/opt 目录，建立一个 mvn 的软链接到/usr/bin 目录中：

```
centos@master:~ $ wget
```

相关网页为 <http://mirror.nexcess.net/apache/maven/maven-3/3.0.5/binaries/apache-maven-3.0.5-bin.tar.gz>。

```
centos@master:~ $ sudo tar -zxvf apache-maven-3.0.5-bin.tar.gz -C /opt/
centos@master:~ $ sudo ln -s /opt/apache-maven-3.0.5/bin/mvn /usr/
bin/mvn
```

使用下面的命令安装其他依赖包：

```
centos@master:~ $ sudo yum install -y Python-devel java-1.7.0-openjdk-
devel zlib-devel libcurl-devel openssl-devel cyrus-sasl-devel cyrus-sasl-
md5 apr-devel subversion-devel
```

## 2. Ubuntu

使用下面的命令来安装在 Ubuntu 操作系统下的所有依赖包：

```
ubuntu@master:~ $ sudo apt-get update
ubuntu@master:~ $ sudo apt-get -y install build-essential openjdk-6-
jdk Python-dev Python-boto libcurl4-openssl-dev libsasl2-dev libapr1-dev libsvn-
dev maven
```

### 9.3.2 构建Mesos

在完成了 Mesos 依赖包的安装之后，我们可以通过下面的步骤构建 Mesos。

(1) 从 Mesos 主站 <http://mesos.apache.org/downloads/> 下载最新的稳定版本。截至本书截稿时，最新版本是 Mesos 0.23.0。

```
ubuntu@master:~$ wget http://www.apache.org/dist/mesos/0.23.0/mesos-
0.23.0.tar.gz
```

(2) 解压 Mesos，删除原有压缩文件，将 Mesos 目录重命名为 mesos，进入目录：

```
ubuntu@master:~ $ tar -xzf mesos-*.tar.gz
ubuntu@master:~ $ rm mesos-*.tar.gz ; mv mesos-* mesos
ubuntu@master:~ $ cd mesos
```

(3) 在 Mesos 目录中创建一个 build 目录，在编译过程中输出的文件将全部放入该目录中，在一次编译完成后，build 中的输出文件可以直接在其他同类型 OS 上使用，无须再在其他节点重新编译。

```
ubuntu@master:~/mesos $ mkdir build
ubuntu@master:~/mesos $ cd build
```

(4) 使用 `configure` 脚本进行配置安装：

```
ubuntu@master:~/mesos/build $ ../configure
```

`configure` 脚本支持对构建环境的调整，使用 `--help` 来查看具体的相关选项。如果相关依赖存在问题，则在 `configure` 脚本中会输出报错信息，我们可以对这些依赖项进行重新安装，一旦配置成功，即可进入下一步。

(5) 接下来使用 `make` 进行编译，`make check` 构建实例 Framework 来验证编译环境，最后通过 `make install` 安装：

```
ubuntu@master:~/mesos/build $ make
ubuntu@master:~/mesos/build $ make check
ubuntu@master:~/mesos/build $ make install
```

### 9.3.3 启动 Mesos

在启动 Mesos 之前，先建立 Mesos 工作目录，并授予权限：

```
ubuntu@master:~ $ sudo mkdir -p /var/lib/mesos
ubuntu@master:~ $ sudo chown `whoami` /var/lib/mesos
```

进入 Mesos build 下的 `bin` 目录中，启动 Mesos Master：

```
ubuntu@master: ~ $ cd /yuhe/mesos-0.23.0/build/bin
ubuntu@master: ~ $ ./mesos-master.sh --work_dir=/var/lib/mesos
```

```
I0815 00:25:30.499855 9657 main.cpp:181] Build: 2015-08-08 15:36:39 by
ubuntu
I0815 00:25:30.502177 9657 main.cpp:183] Version: 0.23.0
I0815 00:25:30.502604 9657 main.cpp:204] Using 'HierarchicalDRF'
allocator
I0815 00:25:30.509304 9657 leveldb.cpp:176] Opened db in 6.183988ms
I0815 00:25:30.514302 9657 leveldb.cpp:183] Compacted db in 4.634415ms
I0815 00:25:30.514549 9657 leveldb.cpp:198] Created db iterator in
50985ns
I0815 00:25:30.514701 9657 leveldb.cpp:204] Searched to beginning of db
in 16883ns
I0815 00:25:30.515071 9657 leveldb.cpp:273] Iterated through 3 keys in
the db in 332165ns
I0815 00:25:30.515434 9657 replica.cpp:744] Replica recovered with log
positions 29 -> 30 with 0 holes and 0 unlearned
I0815 00:25:30.518609 9677 recover.cpp:449] Starting replica recovery
I0815 00:25:30.519467 9677 recover.cpp:475] Replica is in VOTING status
I0815 00:25:30.520191 9677 recover.cpp:464] Recover process terminated
I0815 00:25:30.524001 9657 main.cpp:383] Starting Mesos master
... ..
I0815 00:25:30.533371 9676 master.cpp:368] Master 20150815-002530-
2155456704- 5050-9657 (master) started on 192.168.121.128:5050
```

```

I0815 00:25:30.534142 9676 master.cpp:417] Master allowing unauthenticated
frameworks to register
I0815 00:25:30.534281 9676 master.cpp:422] Master allowing unauthenticated
slaves to register
I0815 00:25:30.534350 9676 master.cpp:459] Using default 'crammd5'
authenticator
W0815 00:25:30.534494 9676 authenticator.cpp:504] No credentials provided,
authentication requests will be refused.
I0815 00:25:30.534611 9676 authenticator.cpp:511] Initializing server
SASL
I0815 00:25:30.550974 9676 master.cpp:1481] The newly elected leader is
master@192.168.121.128:5050 with id 20150815-002530-2155456704-5050-9657
I0815 00:25:30.551311 9676 master.cpp:1494] Elected as the leading
master!
... ..

```

#### 启动 Mesos Slave:

```

ubuntu@master: ~ $ cd /yuhe/mesos-0.23.0/build/bin
ubuntu@master: ~ $ ./mesos-slave.sh --master=master:5050

I0815 00:29:19.930410 9713 main.cpp:162] Build: 2015-08-08 15:36:39
by ubuntu
I0815 00:29:19.933251 9713 main.cpp:164] Version: 0.23.0
I0815 00:29:19.955884 9713 containerizer.cpp:111] Using isolation:
posix/cpu,posix/mem
I0815 00:29:19.981819 9713 main.cpp:249] Starting Mesos slave
I0815 00:29:19.989830 9731 slave.cpp:190] Slave started on 1)@192.168.
121.128:5051
... ..
I0815 00:29:19.992074 9731 slave.cpp:354] Slave resources: cpus(*):1;
mem(*):997; disk(*):4471; ports(*):[31000-32000]
I0815 00:29:19.992539 9731 slave.cpp:384] Slave hostname: master
I0815 00:29:19.992722 9731 slave.cpp:389] Slave checkpoint: true
I0815 00:29:20.034409 9735 state.cpp:36] Recovering state from '/tmp/
mesos/meta'
I0815 00:29:20.041765 9733 status_update_manager.cpp:202] Recovering
status update manager
I0815 00:29:20.048670 9732 containerizer.cpp:316] Recovering containerizer
I0815 00:29:20.080144 9733 slave.cpp:4026] Finished recovery
I0815 00:29:20.081477 9733 slave.cpp:4179] Querying resource estimator
for oversubscribable resources
I0815 00:29:20.082921 9733 slave.cpp:684] New master detected at
master@192.168.121.128:5050
I0815 00:29:20.083400 9735 status_update_manager.cpp:176] Pausing sending
status updates
I0815 00:29:20.087071 9733 slave.cpp:709] No credentials provided.
Attempting to register without authentication
I0815 00:29:20.087558 9733 slave.cpp:720] Detecting new master
I0815 00:29:20.088213 9733 slave.cpp:4193] Received oversubscribable

```

## PaaS 实现与运维管理：基于 Mesos + Docker + ELK 的实战指南

```
resources from the resource estimator
I0815 00:29:20.925995 9732 slave.cpp:859] Registered with master
master@192.168.121.128:5050; given slave ID 20150815-002530-2155456704-
5050-9657-S0
I0815 00:29:20.926450 9733 status_update_manager.cpp:183] Resuming sending
status updates
I0815 00:29:20.927486 9732 slave.cpp:918] Forwarding total oversubscribed
resources
I0815 00:29:35.090006 9734 slave.cpp:4179] Querying resource estimator for
oversubscribable resources
I0815 00:29:35.091192 9734 slave.cpp:4193] Received oversubscribable
resources from the resource estimator
```

Mesos includes various example test frameworks written in C++, Java, and Python.

They can be used to verify that the cluster is configured properly. The following test

framework is written in C++, and it runs five sample applications. We will run it using

the following command:

在 Mesos 的 build 目录下包含了用 C++、Java 和 Python 所写的测试框架，它能够用来验证所安装的 Mesos 环境是否正常。下面我们使用 C++ 测试框架运行 5 个示例程序，通过下面的命令执行：

```
ubuntu@master: ~ $ cd /yuhe/mesos-0.23.0/build
ubuntu@master: ~ $ ./src/test-framework --master=master:5050

I0815 00:34:37.166911 9788 sched.cpp:157] Version: 0.23.0
I0815 00:34:37.176105 9804 sched.cpp:254] New master detected at master@192.
168.121.128:5050
I0815 00:34:37.178732 9804 sched.cpp:264] No credentials provided.
Attempting to register without authentication
I0815 00:34:37.190166 9804 sched.cpp:448] Framework registered with
20150815-002530-2155456704-5050-9657-0000
Registered!
Received offer 20150815-002530-2155456704-5050-9657-00 with cpus(*):1;
mem(*):997; disk(*):4471; ports(*):[31000-32000]
Launching task 0 using offer 20150815-002530-2155456704-5050-9657-00
Received offer 20150815-002530-2155456704-5050-9657-01 with mem(*):869;
disk(*):4471; ports(*):[31000-32000]
Task 0 is in state TASK_RUNNING
Task 0 is in state TASK_FINISHED
Received offer 20150815-002530-2155456704-5050-9657-02 with cpus(*):1;
mem(*):997; disk(*):4471; ports(*):[31000-32000]
Launching task 1 using offer 20150815-002530-2155456704-5050-9657-02
Task 1 is in state TASK_RUNNING
Task 1 is in state TASK_FINISHED
Received offer 20150815-002530-2155456704-5050-9657-03 with cpus(*):1;
```

```

mem(*):997; disk(*):4471; ports(*):[31000-32000]
  Launching task 2 using offer 20150815-002530-2155456704-5050-9657-03
  Task 2 is in state TASK_RUNNING
  Task 2 is in state TASK_FINISHED
  Received offer 20150815-002530-2155456704-5050-9657-04 with cpus(*):1;
mem(*):997; disk(*):4471; ports(*):[31000-32000]
  Launching task 3 using offer 20150815-002530-2155456704-5050-9657-04
  Task 3 is in state TASK_RUNNING
  Task 3 is in state TASK_FINISHED
  Received offer 20150815-002530-2155456704-5050-9657-05 with cpus(*):1;
mem(*):997; disk(*):4471; ports(*):[31000-32000]
  Launching task 4 using offer 20150815-002530-2155456704-5050-9657-05
  Task 4 is in state TASK_RUNNING
  Task 4 is in state TASK_FINISHED
  I0815 00:34:46.455538 9805 sched.cpp:1591] Asked to stop the driver
  I0815 00:34:46.455638 9805 sched.cpp:835] Stopping framework '20150815-
002530-2155456704-5050-9657-0000'
  I0815 00:34:46.456338 9788 sched.cpp:1591] Asked to stop the driver

```

上面的日志信息显示 Framework 向 Master 进行注册，接收其提供的计算资源，并运行了 5 个 task 后退出执行。下面我们看看 Python 测试框架，它以下的命令运行：

```

ubuntu@master: ~ $ cd /yuhe/mesos-0.23.0/build
ubuntu@master: ~ $ ./src/examples/Python/test-framework master:5050
I0815 00:41:02.361846 10022 sched.cpp:157] Version: 0.23.0
I0815 00:41:02.366164 10032 sched.cpp:254] New master detected at
master@192.168.121.128:5050
I0815 00:41:02.367384 10032 sched.cpp:264] No credentials provided.
Attempting to register without authentication
I0815 00:41:02.373347 10032 sched.cpp:448] Framework registered with
20150815-003522-2155456704-5050-9841-0001
Registered with framework ID 20150815-003522-2155456704-5050-9841-0001
Received offer 20150815-003522-2155456704-5050-9841-01 with cpus: 1.0
and mem: 997.0
  Launching task 0 using offer 20150815-003522-2155456704-5050-9841-01
  Task 0 is in state TASK_RUNNING
  Task 0 is in state TASK_FINISHED
  Received message: 'data with a \x00 byte'
  Received offer 20150815-003522-2155456704-5050-9841-02 with cpus: 1.0
and mem: 997.0
  Launching task 1 using offer 20150815-003522-2155456704-5050-9841-02
  Task 1 is in state TASK_RUNNING
  Task 1 is in state TASK_FINISHED
  Received message: 'data with a \x00 byte'
  Received offer 20150815-003522-2155456704-5050-9841-03 with cpus: 1.0
and mem: 997.0
  Launching task 2 using offer 20150815-003522-2155456704-5050-9841-03
  Task 2 is in state TASK_RUNNING
  Task 2 is in state TASK_FINISHED

```

```
Received message: 'data with a \x00 byte'
Received offer 20150815-003522-2155456704-5050-9841-04 with cpus: 1.0
and mem: 997.0
Launching task 3 using offer 20150815-003522-2155456704-5050-9841-04
Task 3 is in state TASK_RUNNING
Task 3 is in state TASK_FINISHED
Received message: 'data with a \x00 byte'
Received offer 20150815-003522-2155456704-5050-9841-05 with cpus: 1.0
and mem: 997.0
Launching task 4 using offer 20150815-003522-2155456704-5050-9841-05
Task 4 is in state TASK_RUNNING
Task 4 is in state TASK_FINISHED
All tasks done, waiting for final framework message
Received message: 'data with a \x00 byte'
All tasks done, and all messages received, exiting
I0815 00:41:06.709058 10031 sched.cpp:1591] Asked to stop the driver
I0815 00:41:06.709097 10031 sched.cpp:835] Stopping framework
'20150815-003522-2155456704-5050-9841-0001'
I0815 00:41:06.709244 10022 sched.cpp:1591] Asked to stop the driver
```

类似地，我们也能够看到其与 Master 的交互及最后的执行退出情况。

### 9.3.4 高可用 Mesos

服务高可用对于现代数据中心来说非常重要，它要保证在异常时系统功能可快速恢复，并继续提供服务。Mesos Slave 一开始就引入了避免单点故障的架构，面向最常见的三种场景：机器故障、进程异常及版本升级。Mesos 在 Master、Slave 与 Framework 三个组件上提供了高可用策略。

如果在 Slave 节点发生机器故障，则 Master 将通过健康检查发现，随后通知 Framework 关于 Slave 异常的信息。对 Framework 而言，它可以依据这些信息选择重新调度，将任务运行到其他 Slave 节点，也可以实现自己的健康检查方式，直接对 Slave 节点上的执行任务进行判断。在机器恢复后，Slave 进程重新启动，它会重新注册到 Master 中成为可用节点。对于 Slave 进程异常或者版本升级的情况，在 Slave 节点上的执行器及任务并不会受到影响，在 Slave 进程恢复后，它们会通过相关信息恢复到之前的任务管理状态。

对于 Framework 而言，无论是机器故障、进程异常还是版本升级，它们并不会影响到 Slave 节点上的任务执行。Framework 可以实现自己的高可用，保证在发生异常后快速地重新注册到 Master 中，随后在 Slavemaster 上获取所有任务的状态信息。

Master 通过 Leader 选举来保证其高可用性，当 Active Master 的机器故障、进程异常或者是版本升级时，Standby Master 能很快选举出新的 Leader，重新接管整个集群。所有 Framework、Slave 及其上运行的任务都不会受到影响。Framework 和 Slave 会重新注册到新 Master 上来。集群中所有的任务状态信息在 Master 上是有保存的，当 Master 的多个实例在



运行时，它们会同步这部分数据，在每个 Master 节点上并不会保存全部信息。Master 使用 ZooKeeper 分布式协调系统来实现选举，保证高可用。

下面我们构建一个高可用的 Mesos 集群，在集群中有三个 OS，分别是 master、slave1、slave2，三台机器上的 Mesos 安装了文件并进行了同步。我们运行三个 Master，启动时连入 ZooKeeper 用作选举发现，推举出一个 Active Master，其他两个作为 Standby。

### 1. 启动 Master

```
ubuntu@master: ~ $ cd /yuhe/mesos-0.23.0/build
ubuntu@master: ~ $ nohup ./bin/mesos-master.sh --ip=`hostname -i` \ --
zk=zk://192.168.121.128:2181,192.168.121.128:2182,192.168.121.128:2183/mes
os --quorum=2 --external_log_file==/var/lib/mesos/log --cluster=paas -
-work_dir=/var/lib/mesos >/var/lib/mesos/master/nohup.out &
```

```
ubuntu@slave1: ~ $ cd /yuhe/mesos-0.23.0/build
ubuntu@slave1: ~ $ nohup ./bin/mesos-master.sh --ip=`hostname -i` \ --
zk=zk://192.168.121.128:2181,192.168.121.128:2182,192.168.121.128:2183/mes
os --quorum=2 --external_log_file==/var/lib/mesos/log --cluster=paas -
-work_dir=/var/lib/mesos >/var/lib/mesos/master/nohup.out &
```

```
ubuntu@slave2: ~ $ cd /yuhe/mesos-0.23.0/build
ubuntu@slave2: ~ $ nohup ./bin/mesos-master.sh --ip=`hostname -i` \ --
zk=zk://192.168.121.128:2181,192.168.121.128:2182,192.168.121.128:2183/mes
os --quorum=2 --external_log_file==/var/lib/mesos/log --cluster=paas -
-work_dir=/var/lib/mesos >/var/lib/mesos/master/nohup.out &
```

Master 启动时带了一个 quorum 的参数，该参数决定了集群中参与选举的大多数值，集群中所有 Master 总节点数、Quorum 数及容错数存在如表 9-1 所示的关系。

表 9-1 Mesos 集群数据

Masters 总数	Quorum 数	容 错 数
1	1	0
3	2	1
5	3	2
...	...	...
2N - 1	N	N - 1

官方推荐的高可用架构下的 Master 节点数是 3 个或者 5 个。Quorum 数并不仅仅代表参与选举的总投票数，其还关系到整个集群中的 Framework、Slave、Task 等数据状态持久化后的复制份数。之前已提到，在 Master 节点上会持久化这些状态信息，这些内容放在了 replicated\_log 文件夹中，这些状态数据在高可用场景下是依据 Quorum 数分散保存到所有

Master 节点中的，因此 Master 集群中允许出错的节点数是受控的，例如在 5 个 Master 节点中只允许两个节点异常，这与状态文件持久化有关。

在总节点数、Quorum 数固定（例如总数为 5，Quorum 为 3）的情况下，新加入一个 Master 会导致整个集群中的状态信息出现异常，因此在新增、删除节点时要特别注意。在未来的版本中，Mesos 会加入白名单机制对这种情况进行防护。

### 2. 新增 Quorum

在一个固定的 Master 高可用架构下，我们可能会需要新增 Master 节点来提升其容错性，需要特别注意的是要保证其执行次序。下面是一个 3 节点扩容到 5 节点的场景的示例，Quorum 将由 2 变成 3。

(1) 最开始 3 个 Master 节点以 “--quorum=2” 运行。

(2) 以 “--quorum=3” 重启最初的 3 个 Master 节点。

(3) 加入新增的两个节点，以 “--quorum=3” 运行。

新增 Quorum 时要遵循的原则是先调大 Quorum，再加入 Master 节点。

由 1 个节点扩容到 3 个时，即由单节点扩充到高可用架构时，需要清理掉整个 replicated\_log 目录的内容，之后进行重启。

(1) 停止单一节点 Master。

(2) 删除 replicated 日志目录的数据（replicated\_log 目录在工作目录下）。

(3) 启动原 Master 节点及新增的两个节点，将 Quorum 调整为 2。

### 3. 减少 Quorum

减少 Quorum 的步骤与新增正好相反，首先要移除 Master 节点，之后再调整 Quorum。下面的步骤由 5 个 Master 节点调整到 3 个，其 Quorum 值由 3 调整为 2。

(1) 一开始有 5 个 Master 节点在运行，其 Quorum 值为 3。

(2) 移除两个 Master 节点，保证其不会再重启且加入到集群，现在剩余 3 个节点在运行，Quorum 值为 3。

(3) 重启剩余的两个节点，以 “--quorum=2” 运行。

### 4. 启动 Slave

接下来，我们分别在三台机器上启动 Slave 节点，这次的 Master 地址是 ZooKeeper 地址：

```
ubuntu@master: ~ $ cd /yuhe/mesos-0.23.0/build
ubuntu@master: ~ $ nohup ./bin/mesos-slave.sh --master=zk://192.168.121.128:2181,192.168.121.128:2182,192.168.121.128:2183/mesos --containerizers=Docker,mesos --work_dir=/var/lib/mesos > /var/lib/mesos/slave/nohup.out &
```

```

ubuntu@slave1: ~ $ cd /yuhe/mesos-0.23.0/build
ubuntu@ slave1: ~ $ nohup ./bin/mesos-slave.sh --master=zk://192.168.121.128:2181,192.168.121.128:2182,192.168.121.128:2183/mesos --containerizers=Docker,mesos --work_dir=/var/lib/mesos > /var/lib/mesos/slave/nohup.out &

ubuntu@ slave2: ~ $ cd /yuhe/mesos-0.23.0/build
ubuntu@ slave2: ~ $ nohup ./bin/mesos-slave.sh --master=zk://192.168.121.128:2181,192.168.121.128:2182,192.168.121.128:2183/mesos --containerizers=Docker,mesos --work_dir=/var/lib/mesos > /var/lib/mesos/slave/nohup.out &

```

Mesos 使用 ZooKeeper 的 `contender` 和 `detector` 抽象接口实现了 Leader 的选举与探测。`contender` 参与到 Leader 的竞选中，`detector` 用于探测当前集群中谁是 Leader。Master 对两个接口都有实现，不仅要查询谁是 Leader，还要参与选举。而 Slave 及 Framework 主要用于探测当前集群中谁是真正的 Master。

在 Master、Slave 及 Framework 之间，ZooKeeper 实际上成了高可用通信中心，在 ZooKeeper 上的连接建立、重建、session 超时，以及 `znode` 节点的创建、删除与更新都会通知到相关 Mesos 中。

Mesos 在与 ZooKeeper 通信时设置了两个超时参数，`contender` 和 `detector` 分别使用 `MASTER_CONTENDER_ZK_SESSION_TIMEOUT`、`MASTER_DETECTOR_ZK_SESSION_TIMEOUT`，不同的 Mesos 组件在与 ZooKeeper 失去连接时会引发不同的行为。

- 如果 Slave 与 ZooKeeper 失去连接，则对于 SlaveMaster 发送过来的消息，它会接收但并不执行具体动作，这样做的目的在于防止这些消息来自一个已经不是 Leader 的 Master。当 Slave 与 ZooKeeper 恢复通信后，它将开始重新响应来自 Master 的消息。
- 如果 Framework 调度器与 ZooKeeper 失去了通信，则不同的 Framework 将实现自己的处理逻辑，它关键要考虑的是如何对当前的 Master 保持信任。
- 当 Master 与 ZooKeeper 之间失去了通信时，如果该 Master 是 Leader，那么它将退出 Leader 状态，等待管理员重启恢复。如果其本来就是 Standby Master，则它将不做任何动作，等待与 ZooKeeper 恢复通信。

## 5. Mesos Web UI

Mesos 提供 Web UI 界面来查询关于集群的信息报告，我们可以通过 `<master-host>:<port>` 来对页面进行访问，如图 9-5、图 9-6 所示是打开的页面，在其菜单栏上对 Mesos 的 Framework、Slave 等都进行了展示。

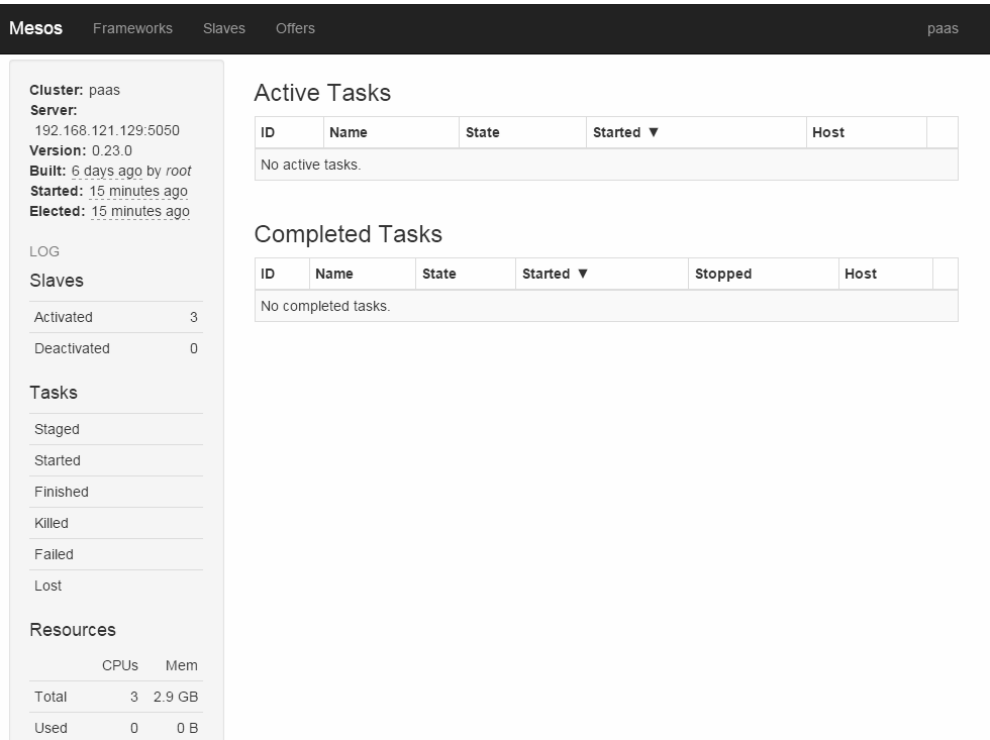


图 9-5 Mesos Web UI 主页

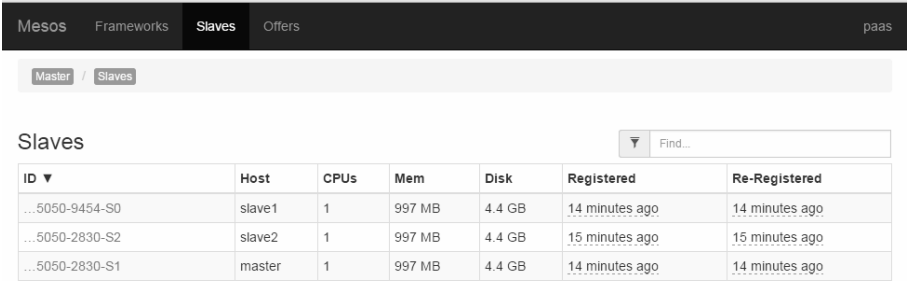


图 9-6 Mesos Web UI Slaves 页

### 9.3.5 Slave移除限速

Slave 通过 ZooKeeper 确认集群中 Master 的健康状态，Master 则会直接对其下的所有 Slave 进行定期的健康检查。Master 发现 Slave 出现异常时，它会将 Slave 节点标注为 deactivated 状态，并通知 Framework。

在 Mesos 工作流中我们提到，Slave 的 manager 进程出现异常并不代表其上的任务有问题。对 Master 而言，它不会干涉这些任务，它的职责是资源管理，因此它标注异常 Slave，通知 Framework 即真正的任务调度者来决策如何处理。

假设有这样一种场景，当出现网络分区异常或者 Master 进程自身 Bug 异常，对 Slave 健康检查无法完成时，它会将所有 Slave 标识为 deactivated，并通知 Framework 处理，这很可能会导致集群下所有正常的工作瘫痪。Slave 移除限速的引入是为了防止这种情况出现，它至少保证了让运维人员有时间通过监控或者日志来判断是否是 Master 异常，以免造成大的影响。在 Mesos 中支持两种类型的限速配置。

- 百分比控制 Slave 移除。通过--recovery\_slave\_removal\_limit 参数进行设置，如果设置成 10，则代表当有 10%的 Slave 出现异常，Slave 注册状态移除时，若再有 Slave 出现问题，则 Master 将不做移除处理，而是等待运维人员进行判断。默认值为 100，其代表不做百分比限速控制。
- 时间限速控制 Slave 移除。通过--slave\_removal\_rate\_limit 参数进行设置，假如设置成 5/60，则它表示 60 分钟内只允许 5 个 Slave 节点被移除。

## 9.4 Mesos运维

### 9.4.1 认证管理

一般的权限控制体系有三个重要概念：用户、组与权限。用户属于零或多个组，权限可以赋予用户，也可以赋予组。如果用户属于一个组，那么他将拥有该组下的相关权限。

Mesos 有着自己的权限体系概念，我们与标准的用户、组、权限匹配来帮助我们快速理解。

Framework 向 Master 注册时，它可以提交 principal、role 与 user 三个值：principal 相当于用户；role 代表其在 Mesos 集群中扮演的角色，匹配到组；而 user 指 Framework 启动任务所用的系统用户。

Mesos 对 Framework 提交过来的信息进行了权限控制，其有以下三种行为。

#### 1) register\_frameworks

在 Framework 注册或者重新注册到 Master 时，Master 会依照这个配置判断 principal 可以分配什么 role，不能分配什么 role。

#### 2) run\_tasks

run\_tasks 将检查是否允许 Framework 以 user 的角色运行 executor/task，这个 user 是系统用户。

#### 3) shutdown\_frameworks

即指定的 principal 是否允许停止一个 Framework。

在以上动作中涉及两个对象（object）：一个是 user，一个是 role。Mesos 的权限控制体系有着自己的概念：principal、action 及 object。

权限控制内容以 JSON 格式体现，在 Master 命令行启动时，通过设置--acls 参数来指定 JSON 内容或者一个文件。

下面来看一些权限控制的例子。

(1) framework foo 和 bar 能够以 alice 用户的角色运行任务：

```
{
  "run_tasks": [
    {
      "principals": { "values": [ "foo", "bar" ] },
      "users": { "values": [ "alice" ] }
    }
  ]
}
```

(2) 所有 Framework 都可以以 guest 用户的角色运行任务：

```
{
  "run_tasks": [
    {
      "principals": { "type": "ANY" },
      "users": { "values": [ "guest" ] }
    }
  ]
}
```

(3) 没有用户可以以 root 角色运行任务：

```
{
  "run_tasks": [
    {
      "principals": { "type": "NONE" },
      "users": { "values": [ "root" ] }
    }
  ]
}
```

(4) Framework foo 允许以 analytics、ads 角色注册，后续可以将资源以独享或者权重的方式分配给这些角色：

```
{
  "register_frameworks": [
    {
      "principals": { "values": [ "foo" ] },
      "roles": { "values": [ "
analytics", "ads" ] }
    }
  ]
}
```

```
    ]
  }
}
```

(5) foo 允许以 analytics 角色注册，其他用户都不能用此用户的角色注册：

```
{
  "register_frameworks": [
    {
      "principals": { "values": [ "foo" ] },
      "roles": { "values": [ "analytics" ] }
    },
    {
      "principals": { "type": "NONE" },
      "roles": { "values": [ "analytics" ] }
    }
  ]
}
```

(6) ops principal 可通过 ‘/teardown’ HTTP 请求停止框架：

```
{
  "permissive" : false,

  "shutdown_frameworks": [
    {
      "principals": { "values": [ "ops" ] },
      "framework_principals": { "type": "
ANY " }
    }
  ]
}
```

Mesos 组件之间的通信机制是由第三方库文件 libprocess 实现的，它们之间的通信协议类似于 HTTP。Mesos 在 0.23.0 版本之前的通信内容都是明文，在 0.23.0 版本中引入了 SSL 证书加密功能。

在使用证书加密功能前，先要安装 openssl、libevent 两个依赖包，对于 libevent 建议采用其 2.0.22-stable 稳定版本。

我们需要重新编译 Mesos，在 configure 中加入如下选项：

```
../configure --enable-libevent --enable-ssl
```

我们一旦成功地构建与安装后，就需要在 Master、Salve、Framework 及其他 libprocess 库文件的进程中设置以下环境变量来使用证书功能。

1) SSL\_ENABLED=(false|0,true|1) [default=false|0]

打开或关闭 SSL，其默认是关闭的，在选择打开时，出去的连接内容都将通过 SSL 协议进行加密，而进来的连接内容将期望通过 SSL 协议加密。在设置该环境变量后，后续的

其他环境变量才有效。

2) `SSL_SUPPORT_DOWNGRADE=(false|0,true|1) [default=false|0]`

控制是否允许建立非 SSL 连接。这个设置用于表示接收端在收到非 SSL 的请求后，是否愿意降级，采用非 SSL 协议进行通信。

3) `SSL_CERT_FILE=(path to certificate)`

指定本地证书文件所在路径。

4) `SSL_KEY_FILE=(path to key)`

指定 OpenSSL 私钥所在的路径。

5) `SSL_VERIFY_CERT=(false|0,true|1) [default=false|0]`

指定是否需要对证书进行验证，默认为 false，即使有相关证书，也无须进行验证。

6) `SSL_REQUIRE_CERT=(false|0,true|1) [default=false|0]`

与 `SSL_VERIFY_CERT` 不同的是，`SSL_REQUIRE_CERT` 将要求所有的连接在建立时都需要进行证书验证，只有通过后才允许建立连接。

7) `SSL_VERIFY_DEPTH=(N) [default=4]`

用于证书验证的最大路径深度。

8) `SSL_CA_DIR=(path to CA directory)`

用于自动发现 CA 证书的路径。

9) `SSL_CA_FILE=(path to CA file)`

用于指定具体的 CA 文件。

10) `SSL_ENABLE_SSL_V3=(false|0,true|1) [default=false|0]`

11) `SSL_ENABLE_TLS_V1_0=(false|0,true|1) [default=false|0]`

● `SSL_ENABLE_TLS_V1_1=(false|0,true|1) [default=false|0]`

● `SSL_ENABLE_TLS_V1_2=(false|0,true|1) [default=true|1]`

以上环境变量用来在不同的 SSL 协议间进行切换，默认采用 TLS V1.2。

## 9.4.2 监控管理

Mesos 主 Slave 节点提供的可观察的监控指标，报告了一组统计数据 and 指标，使我们能够监控资源的使用情况，及早发现异常情况。这些信息包括有关的可用资源，包括



Master、Slave 的活动信息、注册的 Framework 情况、相关任务状态等。我们需要使用这些信息来构建自动报警，并将性能指标绘制到图形界面上。

在监控方面，Mesos 提供了两种不同类型的衡量标准：计数与求值。

- 计数：跟踪离散的事件，并单调递增。这种类型的度量的值始终是一个自然数，计算出失败的 task 的数量和 Slave 注册的数量。计数给出的结果并不能直观地反映问题所在，需要我们设定频率，观察前、后两次计数值的变化。
- 求值：表示一些指标的瞬时样本。例如在集群中使用的内存量和连接数等。这种类型的一些度量会直观地反映当前情况。对于监控来说，我们需要确定的是该值是否过高、或者过低，在持续多长时间后任务异常。

### 1. Master 节点监控

我们通过下面的 URL 来获取 Master 的监控指标：

`http://<mesos-master-ip>:5050/metrics/snapshot`

响应结果是一个键-值对的 JSON 文本对象，其分别是监控指标名称和其对应的值。

如表 9-2 所示的数据提供了有关集群中可用的总资源及其当前使用情况的信息。高资源使用率的时间持续期可能表示要增加容量到集群中。

表 9-2 Mesos 资源指标

指 标 名	描 述	类 型
master/cpus_percent	分配的 CPU 的百分比	求值
master/cpus_used	分配的 CPU 的数量	求值
master/cpus_total	CPU 的数量	求值
master/disk_percent	分配的磁盘空间的百分比	求值
master/disk_used	以 MB 为单位分配的磁盘空间	求值
master/disk_total	以 MB 为单位的磁盘空间	求值
master/mem_percent	分配内存的百分比	求值
master/mem_used	以 MB 为单位分配的内存	求值
master/mem_total	以 MB 为单位的内存	求值

如表 9-3 所示的数据提供了有关 Master 目前是否当选，并已运行了多长时间的信息。如果所有的 Master 都没有选举上，则很可能代表 ZooKeeper 出现了问题或者 Master 发生了频繁切换。低运行时间值表示 Master 最近重新启动过。

表 9-3 Mesos 当选信息

指 标 名	描 述	类 型
master/elected	是否当选 Master	求值
master/uptime_secs	运行时间（秒）	求值

如表 9-4 所示的数据提供了 Master 节点自身的系统资源使用情况。Master 节点长时间维持在一个高的资源使用率上，会影响到集群的整体性能。

表 9-4 Mesos Master 资源使用情况的关注值

指 标 名	描 述	类 型
system/cpus_total	这个 Master 节点可用的 CPU 数量	求值
system/load_15min	过去 15 分钟内的平均负载	求值
system/load_5min	过去 5 分钟内的平均负载	求值
system/load_1min	过去 1 分钟内的平均负载	求值
system/mem_free_bytes	空闲内存字节	求值
system/mem_total_bytes	总内存字节	求值

如表 9-5 所示的数据提供了有关 Slave 的事件、数量及状态。通过对比 Slave 之间的数据，某些数量变化小的 Slave 很可能处于不健康状态。

表 9-5 Mesos Master 资源使用情况的关注值

指 标 名	描 述	类 型
master/slave_registrations	Slave 注册 Master 的次数，不包括重注册	计数
master/slave_removals	Slave Master 移除的次数	计数
master/slave_reregistrations	Slave 重注册的次数	计数
master/slave_shutdowns_scheduled	健康检查失败后移除 Slave 的次数，实际上有一些 Slave 并不是真正地被移除，因为 Slave 移除限速，很可能后续又恢复了，但在这里又全部统计上了	计数
master/slave_shutdowns_cancelled	健康检查失败后移除的 Slave，因为限速而没有被马上移除，之后又恢复了与 Master 连接的次数	计数
master/slave_shutdowns_completed	健康检查失败后移除的 Slave 次数，该值代表着真正移除的次数，之后因限速而恢复的次数不包括在内	计数
master/slaves_active	活动的 Slave 数量	求值
master/slaves_connected	已连接的 Slave 数量	求值
master/slaves_disconnected	断开的 Slave 数量	求值
master/slaves_inactive	不活跃的 Slave 数量	求值

如表 9-6 所示的数据提供了有关集群中注册的 Framework 信息。未连接或未活动的 Framework 都表示其存在问题。

表 9-6 Mesos Framework 的信息指标

指 标 名	描 述	类 型
master/frameworks_active	active 的 Framework 数量	求值
master/frameworks_connected	连接的 Framework 数量	求值
master/frameworks_disconnected	断开连接的 Framework 数量	求值
master/frameworks_inactive	未 active 的 Framework 数量	求值
master/outstanding_offers	已对外提供的资源情况	求值

如表 9-7 所示的数据提供了有关活动和终止任务的信息。任务丢失率偏高意味着集群很可能存在某些问题。其中的统计数字与任务的状态相匹配。

表 9-7 Mesos 任务数统计值

指 标 名	描 述	类 型
master/tasks_error	无效的任务数量	计数
master/tasks_failed	失败的任务数量	计数
master/tasks_finished	完成的任务数量	计数
master/tasks_killed	杀死的任务数量	计数
master/tasks_lost	丢失的任务数量	计数
master/tasks_running	正在运行的任务数量	求值
master/tasks_staging	准备阶段的任务数量	求值
master/tasks_starting	起动的任务数量	求值

如表 9-8 所示的数据提供了有关 Master 和 Slave 之间，以及 Framework 和执行器之间的消息情况。消息的丢失率偏高可能代表着网络存在问题。

表 9-8 Mesos 消息统计值

指 标 名	描 述	类 型
master/invalid_framework_to_executor_messages	Framework 到执行器间消息的无效数量	计数
master/invalid_status_update_acknowledgements	无效的状态更新确认数量	计数
master/invalid_status_updates	无效的状态更新数量	计数
master/dropped_messages	丢弃的消息数量	计数
master/messages_authenticate	验证的消息数量	计数
master/messages_deactivate_framework	Framework 停用的消息数量	计数
master/messages_exited_executor	执行器消息的退出数量	计数
master/messages_framework_to_executor	从消息 SlaveFramework 到执行器的数量	计数
master/messages_kill_task	删除任务的消息数量	计数
master/messages_launch_tasks	启动任务的消息数量	计数
master/messages_reconcile_tasks	协调任务的消息数量	计数
master/messages_register_framework	Framework 注册的消息数量	计数
master/messages_register_slave	Slave 注册的消息数量	计数
master/messages_reregister_framework	Framework 重新注册的消息数量	计数
master/messages_reregister_slave	Slave 重新注册的消息数量	计数
master/messages_resource_request	资源请求的消息数量	计数
master/messages_revive_offers	资源接收的消息数量	计数
master/messages_status_udpate	状态更新的消息数量	计数
master/messages_status_update_acknowledgement	状态更新确认消息的数量	计数
master/messages_unregister_framework	Framework 注销的消息数量	计数
master/messages_unregister_slave	Slave 注销的消息数量	计数

续表

指 标 名	描 述	类 型
master/valid_framework_to_executor_messages	Framework 到执行器的有效消息数量	计数
master/valid_status_update_acknowledgements	有效的状态更新确认消息的数量	计数
master/valid_status_updates	有效的状态更新消息的数量	计数

如表 9-9 所示的数据提供了不同类型的事件队列中的相关信息。

表 9-9 Mesos 事件队列信息指标

指 标 名	描 述	类 型
master/event_queue_dispatches	调度的事件队的列数量	求值
master/event_queue_http_requests	HTTP 请求的事件队列的数量	求值
master/event_queue_messages	消息的事件队列的数量	求值

如表 9-10 所示的数据提供了有关读取和写入延时到 Slave 注册者的信息。

表 9-10 Mesos Slave 注册延时指标

指 标 名	描 述	类 型
registrar/state_fetch_ms	注册者的读取延迟，单位是毫秒	求值
registrar/state_store_ms	注册者的写入延迟，单位是毫秒	求值
registrar/state_store_ms/max	注册者的写入最大延迟	求值
registrar/state_store_ms/min	注册者的写入最小延迟	求值
registrar/state_store_ms/P50	注册者的写入平均延迟	求值
registrar/state_store_ms/P90	90%的注册者写入延迟	求值
registrar/state_store_ms/P95	95%的注册者写入延迟	求值
registrar/state_store_ms/P99	99%的注册者写入延迟	求值
registrar/state_store_ms/P999	99.9%的注册者写入延迟	求值
registrar/state_store_ms/p9999	99.99%的注册者写入延迟	求值

下面是一些用来判断集群所存在问题的例子。

- master/uptime\_secs 低：master 已重新启动。
- master/uptime\_secs<60 的情况持续了一段时间：该集群有一个 Master 在不断发生切换。
- master/tasks\_lost 正在迅速增加：集群中的任务正在消失，可能的原因包括硬件故障，在 Framework 中有一个错误，或者在 Master 中有一个错误。对任务的状态了解有助于我们在监控中判断问题所在。
- master/slaves\_active 低：Slave 在连接到 Master 时出现问题。
- master/cpus\_percent>0.9 的情况持续了一段时间：集群的 CPU 利用率接近容量。

- master/mem\_percent>0.9 的情况持续了一段时间：集群存储利用率接近容量。
- master/elected 为 0 的情况持续了一段时间：当前没有 Master 为 active。

## 2. slave 节点监控

我们通过下面的 URL 来获取 Master 的监控指标：

`http://<mesos-slave>:5051/metrics/snapshot`

响应结果同样是一个键-值对的 JSON 文本对象，其分别是监控指标 C 名称和其对应的值。

表 9-11 提供了关于 Slave 可用的总资源及其当前使用情况的信息。

表 9-11 Mesos Slave 资源指标

指 标 名	描 述	类 型
slave/cpus_percent	分配的 CPU 的百分比	求值
slave/cpus_used	分配的 CPU 数量	求值
slave/cpus_total	CPU 数量	求值
slave/disk_percent	分配的磁盘空间的百分比	求值
slave/disk_used	以 MB 为单位分配的磁盘空间	求值
slave/disk_total	以 MB 为单位的磁盘空间	求值
slave/mem_percent	分配内存的百分比	求值
slave/mem_used	以 MB 为单位分配的内存	求值
slave/mem_total	以 MB 为单位的内存	求值

表 9-12 提供了有关 Slave 是否注册到 Master，以及运行了多长时间的信息。

表 9-12 Mesos Slave 注册指标

指 标 名	描 述	类 型
slave/registered	Slave 是否注册到 Master	求值
slave/uptime_secs	运行时间（秒）	求值

表 9-13 提供了有关 Slave 的系统指标。

表 9-13 Mesos Slave 系统指标

指 标 名	描 述	类 型
system/cpus_total	可用的 CPU 数量	求值
system/load_15min	过去 15 分钟内的平均负载	求值
system/load_5min	过去 5 分钟内的平均负载	求值
system/load_1min	平均负载	求值
system/mem_free_bytes	空闲内存的字节	求值
system/mem_total_bytes	总内存的字节	求值

表 9-14 提供了有关 Slave 运行的执行器实例的指标。

表 9-14 Mesos Slave 运行执行器实例的指标

指 标 名	描 述	类 型
slave/frameworks_active	活动的 Framework 数量	求值
slave/executors_registering	执行器的注册数量	求值
slave/executors_running	执行器的运行数量	求值
slave/executors_terminated	执行器终止数量	计数
slave/executors_terminating	执行器正在终止的数量	求值

表 9-15 提供了有关活动和终止的任务指标。

表 9-15 Mesos Slave 活动和终止的任务指标

指 标 名	描 述	类 型
slave/tasks_failed	失败的任务数量	计数
slave/tasks_finished	完成的任务数量	计数
slave/tasks_killed	杀死的任务数量	计数
slave/tasks_lost	丢失的任务数量	计数
slave/tasks_running	正在运行的任务数量	求值
slave/tasks_staging	准备阶段的任务数量	求值
slave/tasks_starting	启动的任务数量	求值

表 9-15 提供了有关 Slave 和它注册的 Master 之间的消息信息指标。

表 9-16 Mesos Slave 和 Master 的消息信息指标

指 标 名	描 述	类 型
slave/invalid_framework_messages	无效的 Framework 消息数量	计数
slave/invalid_status_updates	无效的状态更新数量	计数
slave/valid_framework_messages	有效的 Framework 消息数量	计数
slave/valid_status_updates	有效的状态更新数量	计数

### 9.4.3 容器网络限速

在 Slave 启动的任务中，它们大部分都是以容器的形态运行的，在同一个 OS 上可能存在一个容器将网络资源消耗殆尽的情况，这些资源既可以是网络带宽，也可以是端口数。在一个 OS 上运行多个容器，端口数很可能就成了稀缺资源，对于通过多个连接频繁发起主动请求的应用，它们会占用大量的临时（Ephemeral）端口。

在限制网络资源使用时，同样要对每一个容器的网络资源使用情况进行监控。

默认构建与安装下的 Mesos 并不支持容器网络监控与限制，如果需要加入这部分功

能，则需要引入额外的步骤。

只有 Linux Kernel 3.6 或者以上版本才支持该功能。在安装前需要保证以下库文件已安装：

- libnl3 >= 3.2.26
- iproute >= 2.6.39
- libnl3-devel / libnl3-dev >= 3.2.26

在构建时输入以下命令：

```
$ ./configure --with-network-isolator
$ make
```

在启动 Slave 时，命令行加入下面的项：

```
--isolation="network/port_mapping"
```

在 Linux 操作系统的系统参数中，`sysctl net.ipv4.ip_local_port_range` 用来设置客户端主动访问的随机端口数，其默认是 32768~61000。在启动容器网络监控、限制功能时，会将这个端口范围缩小，而将容器的端口分配职责交由 Slave 负责，以及可控的端口分配。

通过以下命令缩小端口范围：

```
ubuntu@master:~ $ echo "57345 61000" >/proc/sys/net/ipv4/ip_local_port_range
```

编辑/etc/sysctl.conf 文件，修改 `sysctl net.ipv4.ip_local_port_range`，最后通过下面的命令使其生效：

```
ubuntu@master:~ $ sysctl -p
```

现在我们将端口 32768~57344 保留给容器使用，在启动 Slave 的命令行时使用下面的选项：

```
ubuntu@master:~ $ mesos-slave \
--isolation=network/port_mapping \
--resources=ports:[31000-32000];ephemeral_ports:[32768-57344] \
--ephemeral_ports_per_container=1024 \
--egress_rate_limit_per_container=37500KB \
--egress_unique_flow_per_container
```

在启动项中加入了很多新的选项，让我们逐一解读：`--isolation` 表示启动网络限制功能；`--resources` 用来表示容器可用的服务端口、随机端口范围。在上例中，31000~32000 将作为服务端口使用，留给 Master 分配，而 32768~57344 则作为随机端口，在容器中应用发起主动访问使用。`--ephemeral_ports_per_container` 表示每一个容器最大使用的随机端口数；`--egress_rate_limit_per_container` 表示网络带宽限速数，为了保证最低的网络延时，最后可将每一个容器的网络流独立出来，通过设置 `--egress_unique_flow_per_container` 实现。

在 Slave 节点中，通过输入/monitor/statistics.json url 来检查容器的网络使用情况，其会返回每一个容器的网络监控指标，如表 9-17 所示。

表 9-17 Slave 的网络监控指标

指 标 名	描 述	类 型
net_rx_bytes	接收字节数	计数
net_rx_dropped	丢弃接收字节数	计数
net_rx_errors	接收错误数	计数
net_rx_packets	接收包数	计数
net_tx_bytes	传输字节数	计数
net_tx_dropped	丢弃传输字节数	计数
net_tx_errors	传输错误数	计数
net_tx_packets	传输包数	计数

#### 9.4.4 Framework API限速

在多 Framework 的环境下，集群中某些 Framework 对资源的占用会影响到其他 Framework。不同的 Framework 可能具备不同的优先级，有着不同的 SLA。为了避免某些 Framework 向 Master 发送过多的消息信息，占用大量资源，同时保证其他 Framework 的正常运行，Mesos 支持对 Framework API 请求限速。运维人员可以对 Mesos 进行设置，从而控制某一段时间内运行处理的 Framework API 请求数量。在 Master 中有一个内存队列，用于缓存临时超过处理数的请求。当 Framework 超过其请求限制时，它将收到一个 error 信息，这将导致调度工作被放弃，并触发一个专门的错误处理回调函数。

Framework 的限速配置同样是基于 JSON 文档格式的，并有如下三个参数。

- principal: 用来表示一个 Framework 或者一个 Framework 组。
- qps: 每秒能够处理的 API 数量。
- capacity: 该值相当于设置 Master 中的内存队列所允许缓存的最大消息数量。

在 Master 启动命令行中通过--rate-limits 标示指定 JSON 配置文件，加载限速内容。下面是一个示例文件：

```
{
  "limits": [
    {
      "principal": "spark",
      "qps": 55.5,
      "capacity": 100000
    },
    {
      "principal": "marathon",
```



```

    "qps" : 300
  },
  {
    "principal" : "baz",
  }
],
"aggregate_default_qps" : 333,
"aggregate_default_capacity" : 1000000
}

```

在上面的配置文件中三个 principal，分别是 spark、marathon 及 baz。其中 baz 并没有设置 qps 和 capacity，它将采用最下面的 aggregate\_default\_qps、aggregate\_default\_capacity 两个默认设置。

### 9.4.5 Restful接口

在前面的监控部分，我们介绍了如何使用/metric/snapshot 来获取相关的监控指标参数，除了该 URL，Mesos 还提供了其他基于 Restful 接口的 JSON 格式信息，如表 9-18 所示。

表 9-18 监控 URL

URL	说 明
/__processes__	列出集群中的所有进程
/files/browse /files/debug /files/download /files/read	使用 Restful 接口来浏览、下载 Mesos 中的相关文件
logging/toggle	日志模式的设置开关
/master/health	通过 HTTP 状态码来反馈 Master 的健康状态
/master/redirect	重定向到 active Master 上
/master/observe	以列表的格式接收所有服务器的健康状态，包括节点名称、健康等级等
/master/roles	列出当前所分配的角色
/master/state	当前集群中所有组件的状态，包括 Master、Slave 及 Framework
/master/shutdown	通过指定的 Framework ID 停止一个 Framework
/registrar(1)/registry	返回注册者信息
/profiler/start /profiler/stop	停止/启动 Mesos Profiler
/metric/snapshot	提供当前监控状态的指标

在 Mesos 的 Web 界面上提供了详细的 Help 页面来帮助我们使用这些接口，通过访问 <http://master:5050/help> 可找到 REST 接口信息的列表。

如图 9-7 所示是 Mesos 的 Help 页面，它以树形结构展现了所有的 API，通过单击这些节点，我们可以获得更加详细的信息，例如单击 system 节点查看和访问 <http://master:5050/help/system/stats.json>，其结果如图 9-8 所示。

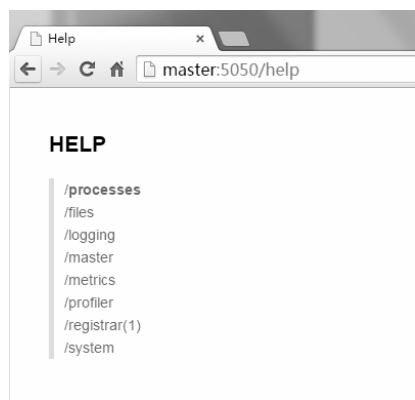


图 9-7 Mesos Web 帮助信息

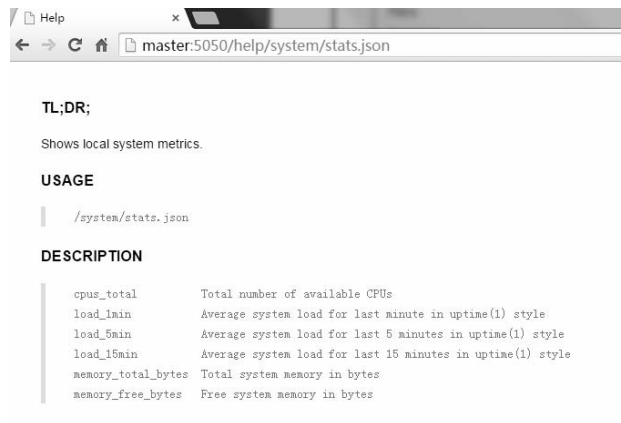


图 9-8 Mesos Web 帮助信息 2

让我们再直接访问 <http://master:5050/system/stats.json>，其中的字段可以参考 Help 页面的说明。如图 9-9 所示。

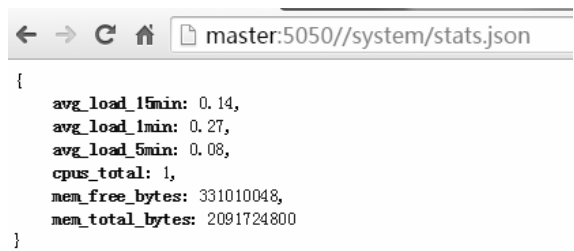


图 9-9 Mesos Web 帮助信息 3

### 9.4.6 配置参数

最新的配置参数在 Mesos 的主站维护（<http://mesos.apache.org/documentation/latest/>）

configuration)。设置 Mesos 配置的地方有两处：首先在环境变量中查找，之后使用启动参数。环境变量与启动参数设置的结果是一样的，环境变量以 MESOS\_前缀为开头，例如--work-dir 选项，在环境变量中设置为 MESOS\_WORK\_DIR。

## 9.5 Mesos资源分配

### 9.5.1 DRF算法

在任何共享的计算机系统中，资源分配是一个关键的构建模块。在这方面已提出的最通用的分配策略是 max-min fairness，它提供了一种能很好地在多用户之间对单资源需求进行分配的机制，最简单的例子是每次分配  $1/N$  的资源给每个用户。这种策略会最大化系统中一个用户收到的最小分配资源，保证每个用户都能够公平地请求资源，给予每个用户一份均等的资源。在 max-min fairness 基础上加入权重以支撑不同的资源分配策略，这些包括优先级、deadline 分配等。鉴于这些特征，不同精确度的加权或非加权 max-min fairness 算法已实现，例如 round-robin、proportional resource sharing、weighted fair queueing。这些算法已经被应用在不同的资源上，包括链路带宽、CPU、内存、存储等。在很多知名的资源调度器上实现了 min-max fairness 分配策略，例如 Hadoop 的 Fair Scheduler、Capacity Scheduler、Quincy scheduler 等。

尽管已经在公平分配上做了大量的工作和实践，人们发现这些策略主要集中在单资源类型的场景下。甚至在多资源类型的环境下，虽然用户有异构资源的请求，典型的资源分配做法却还是使用单类型资源抽象。比如 Hadoop 的公平调度器，在资源分配时使用插槽 (Slot)，Slot 就是对节点资源按照固定大小进行划分而产生的分区。然而事实是集群中不同的作业队 CPU、内存和 I/O 资源有着不同的需求。例如两个  $\langle 1 \text{ CPU}, 3 \text{ GB} \rangle$  and  $\langle 3 \text{ CPU}, 1 \text{ GB} \rangle$  的请求发送给调度器，调度器将怎么比较并保证资源分配的公平呢？

Mesos 的项目团队提出了 Dominant Resource Fairness (DRF，一种通用的多资源 max-min fairness 分配策略)。DRF 背后的直观思想是在多资源环境下一个用户的分配策略应该由用户的 dominant share (主导份额的资源) 决定，dominant share 是指在用户申请的异构资源中，其中一类占比总资源量最多的资源。简而言之，DRF 试图最大化所有用户中的最小 dominant share。举个例子，假如用户 A 运行 CPU 密集的任务而用户 B 运行内存密集的任务，则 DRF 会试图均衡用户 A 的 CPU 资源份额和用户 B 的内存资源份额。在单个资源的情形下，DRF 就会退化为 max-min fairness。

考虑一个总资源量为 9 CPU、18GB RAM 的系统，其中有两个用户，用户 A 运行一个任务的需求向量为 {1CPU, 4GB}，用户 B 运行一个任务的需求向量为 {3CPU, 1GB}。在上述场景中，A 的每个任务消耗总 CPU 的  $1/9$  和总内存的  $2/9$ ，因此 A 的 dominant resource 是

内存；B 的每个任务消耗总 CPU 的 1/3 和总内存的 1/18，因此 B 的 dominant resource 为 CPU。DRF 会均衡用户的 dominant shares，如图 9-10 所示，三个用户 A 的任务总共消耗了 {3CPU, 12GB}，两个用户 B 的任务总共消耗了 {6CPU, 2GB}。在这个分配中，每个用户的 dominant share 是相等的，用户 A 获得了 2/3 的 RAM，而用户 B 获得了 2/3 的 CPU。

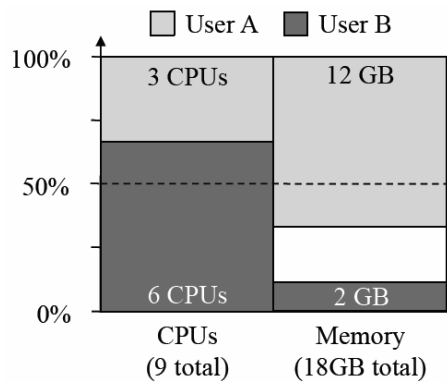


图 9-10 DRF 资源分配的例子

以上分配结果可用下面的方法计算出来： $x$  和  $y$  分别是用户 A 和用户 B 分配任务的数目，用户 A 消耗了  $\{x\text{CPU}, 4x\text{GB}\}$ ，用户 B 消耗了  $\{3y\text{CPU}, y\text{GB}\}$ ，在图 9-10 中用户 A 和用户 B 消耗了同等的 dominant resource；用户 A 的 dominant share 为  $4x/18$ ，用户 B 的 dominant share 为  $3y/9$ ，所以 DRF 分配可以通过求解以下优化问题来得到，如图 9-11 所示。

$$\begin{aligned} &\max(x, y) && \text{(max-min fairness 分配)} \\ &\text{subject to} \\ &\quad x + 3y \leq 9 && \text{(CPU 约束型)} \\ &\quad 4x + y \leq 18 && \text{(MEM 约束型)} \\ &\quad \frac{2x}{9} = \frac{y}{3} && \text{(dominant share 相等)} \end{aligned}$$

图 9-11 DRF 分配数学求解

求解以上问题，可以得出  $x=3, y=2$ ，因而用户 A 获得 {3CPU, 12GB}，B 得到 {6CPU, 2GB}。

DRF 并不总需要使用户的 dominant shares 相等。若一个用户的需求被满足，且用户不需要更多的任务，则多余的资源可以在其他用户之间进行分配，就好像 max-min fairness。DRF 算法需要保证的是每次都从拥有最低 dominant share 的用户中选择一个任务准备运行。此外，如果一类资源被耗尽，则那些不需要这类资源的用户仍然可以继续接收到更多的其他类型的资源。

表 9-19 说明了 DRF 对上面的简单例子的分配过程，在最初 A、B 都没有要运行的任务，DRF 无法判断 A、B 的任务需求量，因此随机选择 A、B 进行运行。DRF 首先选择了 B

来运行一个任务，结果 B 的资源占用率变为  $\{3/9, 1/18\}$ ，B 的 dominant share 变为  $\max\{3/9, 1/18\} = 1/3$ 。接下来 DRF 选择 A，因为此时 A 的 dominant share 为 0。A 的 dominant share 变为  $\max\{1/9, 4/18\} = 2/9$ 。接下来的资源分配依旧是比较二者的 dominant share，因为 A 比 B 小，( $2/9 < 1/3$ )，因此将资源分配给 A。这个过程将持续进行，直到不可能再运行新任务，在这种情况下一般会出现 CPU 饱和。在以上分配过程的最后，用户 A 会得到  $\{3\text{CPU}, 12\text{GB}\}$ ，事件 B 会得到  $\{6\text{CPU}, 2\text{GB}\}$ ，每个用户都获得了  $2/3$  的 dominant resource。以上这个算法在实现时可以用二叉堆来存储每个用户的 dominant share。对于  $n$  个用户，每一次调度决策都消耗了  $O(\log n)$  时间。

表 9-19 DRF 资源分配示例

Schedule	UserA		UserB		CPU total alloc.	RAM total alloc.
	res.shares	dom.shares	res.shares	dom.shares		
UserB	$\langle 0, 0 \rangle$	0	$\langle 3/9, 1/18 \rangle$	1/3	3/9	1/18
UserA	$\langle 1/9, 4/18 \rangle$	2/9	$\langle 3/9, 1/18 \rangle$	1/3	4/9	5/18
UserA	$\langle 2/9, 8/18 \rangle$	4/9	$\langle 3/9, 1/18 \rangle$	1/3	5/9	9/18
UserB	$\langle 2/9, 8/18 \rangle$	4/9	$\langle 6/9, 2/18 \rangle$	2/3	8/9	10/18
UserA	$\langle 3/9, 12/18 \rangle$	2/3	$\langle 6/9, 2/18 \rangle$	2/3	1	14/18

Mesos 的 DRF 重点关注以下两项。

(1) **sharing incentive**: 每个用户都必须更好地共享集群，而不是在集群中专享它们自己的分区。若一个集群具有相同的节点（每个节点都一样）和  $n$  个用户，则一个用户不能在超过  $1/n$  资源的集群分区中分配更多的任务，最多只能分享  $1/n$  的资源。DRF 在原有的单资源 max-min fairness 上引入了对 dominant share 的判断。

(2) **strategy-proofness**: 用户不能从谎报的资源请求中得到好处，不能通过欺骗来提升它的分配。DRF 的分配原则基于用户的使用情况，并通过最小 dominant share 进行分配。

## 9.5.2 DRF权重

DRF 计算每个角色的 dominant share，并按照最小资源需求进行分配。在实际的生产环境运行中，一个公司很少会依据完全公平的策略进行资源分配。例如，一个服务于多个业务公司的 IT 运维机构会按照一定的规则进行资源优先级分配；一个服务于多种计算类型的资源平台会将实时任务与后台任务进行区分，将某些资源调度给实时任务。我们希望按照权重将资源分配给不同的部门、团队或者服务类型。

Mesos 能够通过配置 DRF 权重来解决上面的问题，使用 `--weights` 和 `--roles` 标示来控制权重的分配。`--roles` 用来确认每个 Framework 的角色，而 `--weights` 在 Master 启动时传入一个 role/weight 配对，例如 `role1=weight1, role2=weight2`。

# 服务调度框架 Marathon

## 10.1 Marathon 介绍

### 10.1.1 服务调度平台

Mesos 的核心任务是在多种用户计算类型之间公平地进行资源分配，在其上需要有专属于各类计算类型的 Framework 进行任务调度。传统 IT 企业日常运行最多的是长任务服务型计算，例如应用服务器、Web 服务器、缓存服务器、消息队列等。这些长任务型服务分为包含业务逻辑的应用服务器及不包含业务逻辑的缓存、消息等，统称为中间件。在企业应用架构调整的过程中，不管是面向服务架构（SOA）还是微模块架构（microservice），调整内容都是将应用细粒度拆分到中间件中的方式，以及在架构中引入缓存、队列组件来完成非功能性的需求，例如提升性能、解耦应用等，而作为长任务型服务的中间件变化甚少。在 PaaS 平台中需要一个中间件任务调度平台来完成以下工作。

- 将中间件计算单元发送到基础资源上运行。计算逻辑打包成 Docker 镜像放入仓库，任务调度器向资源管理申请资源后，发送调度命令到指定 Slave 上来获取计算逻辑，并执行任务。
- 对中间件实例运行状态监控，异常主动恢复。调度器充当中间件实例的通用性监控，在出现硬件故障、服务异常时快速在其他节点恢复实例，遵循低耦合、高聚合原则。对于复杂性监控，例如当前活动线程数、等待队列数类监控没有放在调度器中，而是由外部监控平台负责。调度平台要提供 API 接口，当出现容量不足等情况时由外部程序触发行为处理。
- 对中间件任务进行编排，按照组之间的依赖关系启动。一个完整的应用栈包含了 Web 服务器、应用服务器、缓存服务器及后端数据库；而在微模块架构中一个应用由不同业务子模块服务组成，其中可能存在一种服务依赖关系，任务调度可以在将中间件进行分组、分类后，按照依赖关系顺序启动。

Marathon 是 Mesos 上用于负责长任务处理的 Framework，也就是所谓的服务调度框架。Marathon 设计之初的目标是让学生在 Shell 命令中执行的命令都能通过 Marathon 远程调度，并保证长任务进程持续运行。Marathon 能够完成上述命令的调度平台工作，同时具备其他特性。

### 10.1.2 Marathon 实体模型

Marathon 对分布式系统中的应用及其工作流程进行了定义，从全局上看这些实体有利于我们对其实现过程的理解。其相关实体如图 10-1 所示。

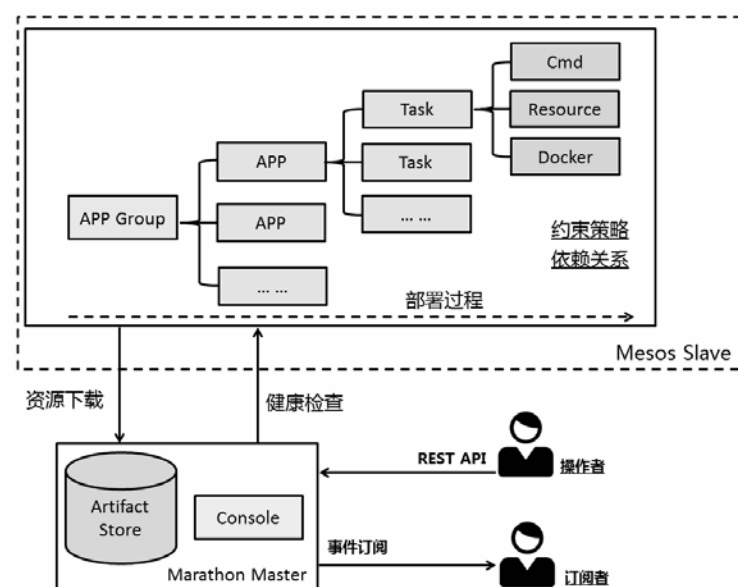


图 10-1 Marathon 实体图

App 即应用，是 Marathon 的中心实体，由多个 Task 构成一个集群，这些 Task 是同构的，运行的逻辑与所需资源是完全一致的。Task 可以是 Shell 命令行代码（Cmd）、远程资源程序（Resource）、Docker 命令（Docker）。

App Group 将多个不同类型的 App 聚合在一起，例如一个应用的全栈包括 Tomcat、Redis、MySQL 等，App Group 将这些内容组合成一个整体。在对 App 进行组合与编排时，Marathon 提供了一些基本策略与规则。

- 约束策略：在 App Group、App 运行时，我们可以定义一些分区执行、资源共享等约束策略，用以提高应用可用性。
- 依赖关系：一个 App Group 的启动、停止存在依赖关系，例如 Tomcat 的启动必须等待 MySQL 先启动，这种关系可以进行定义。

- 部署过程：App Group 或者 App 的启动、停止、扩容等被称为部署过程，我们可以设置回滚、灰度等部署策略，以保证应用的可用性。

在 App 的外部有一个 Marathon Master 作为控制中心存在。它提供 App 所需的外部资源的存储库，也执行一些简单的健康检查任务。除此以外，Marathon 可以提供订阅服务及 REST API 命令集。

- 订阅者：在 Marathon 中所有 App Group 内发生的事件都能够通知到其订阅者，订阅者可以依据这些消息发出额外的动作。
- 操作者：在 Marathon 外部，用户可以通过 REST API 发起控制命令，例如创建、更新、删除、部署 App，操作者可以是实际用户；而对于 PaaS 平台而言，应当在所有 Framework 之前构建一层自己的 API 服务器，对所有操作进行一层封装。

Marathon 中 REST API 是依据实体进行分类的，用户通过 HTTP 向 Marathon 发送请求。下面按类型对 API 进行说明，如表 10-1～表 10-7 所示。

表 10-1 Apps

POST /v2/apps	创建、启动一个新应用
GET /v2/apps	列出所有运行的应用
GET /v2/apps/{appId}	列出 ID 为 appId 的应用
GET /v2/apps/{appId}/versions	列出 ID 为 appId 的应用版本
PUT /v2/apps/{appId}	改变 ID 为 appId 的应用配置
POST /v2/apps/{appId}/restart	轮询重启 ID 为 appId 的应用
DELETE /v2/apps/{appId}	删除应用 appId
GET /v2/apps/{appId}/tasks	列出 appId 的所有任务
DELETE /v2/apps/{appId}/tasks	属于 appId 的所有任务
DELETE /v2/apps/{appId}/tasks/{taskId}	属于 appId 的任务 taskId

表 10-2 App Groups

GET /v2/groups	列出所有的 group
GET /v2/groups/{groupId}	列出 ID 为 groupId 的 group
POST /v2/groups	创建、启动新的 group
PUT /v2/groups/{groupId}	改变 groupId 的配置
DELETE /v2/groups/{groupId}	删除一个 group

表 10-3 Tasks

GET /v2/tasks:	列出所有的 task
POST /v2/tasks/delete:	列表中的所有 task，列表将通过 JSON 文档传送
GET /v2/deployments:	列出所有正在进行的部署
DELETE /v2/deployments/{deploymentId}:	重启或取消 ID 为 deploymentId 的部署动作



表 10-4 Event

POST /v2/eventSubscriptions	订阅事件，将一个 URL 注册成事件接收方
GET /v2/eventSubscriptions	列出所有的事件订阅者
DELETE /v2/eventSubscriptions	取消一个 URL 的事件订阅
GET /v2/events: Attach to the event stream	0.9.0 版本新增的功能，Attach 到事件流中，与 Marathon 服务端建立长连接，后续有任何事件产生，服务端将发送通知

表 10-5 Queue

GET /v2/queue	列出队列中 staging 状态的任务列表
DELETE /v2/queue/{appId}/delay: v0.10.0	0.10.0 版本新增功能，对于有超时限制的 task，当其还在队列中处于 staging 状态时，可以主动将其重置

表 10-6 Server Info

GET /v2/info:	获取关于 Marathon 的信息
GET /v2/leader:	获取 Marathon 集群中的 leader
DELETE /v2/leader:	促使当前 Leader 失效，发起一次选举

表 10-7 其他

GET /ping	为简单的监控，确认服务正常
GET /logging	获取 Marathon 的 logging 状态
GET /help	获取 REST API 的帮助信息
GET /metrics	返回 Marathon 的监控指标值

## 10.2 Marathon使用

### 10.2.1 安装启动

由于 Marathon 会引用 Mesos 相关的动态链接库，所以在安装之前应已安装 Mesos。

下载 Marathon 安装包，当前安装的 Mesos 版本是 0.23.0，Marathon 版本是 0.9.0。

```
ubuntu@master:~ $ wget
http://downloads.mesosphere.com/marathon/v0.9.0/marathon-0.9.0.tgz
```

解压安装包，进入 Marathon 目录中：

```
ubuntu@master:~ $ tar xzf marathon-*.tgz
ubuntu@master:~ $ rm marathon-*.tgz; cd marathon-*
```

设置环境变量，指向 Mesos 动态链接库文件所在的位置：

```
export MESOS_NATIVE_JAVA_LIBRARY= /mesos/build/src/./libs/libmesos.so
```

通过以下的脚本启动 Marathon：

```
./bin/start --master zk://address:2181/mesos --zk zk://address:2181/marathon
```

上面的--master 连入的 ZooKeeper 是 Mesos 所使用的 znode 节点，用来与 Mesos Master 通信，而--zk 是 Marathon 高可用接入的 ZooKeeper，这两个 ZooKeeper 可以是相同集群，也可以是不同的，由此可见，PaaS 中 ZooKeeper 作为分布式协调管理器嵌入在了资源管理、任务调度中。

启动 Marathon 之后可以通过 Web 对其进行访问，如图 10-2 所示。

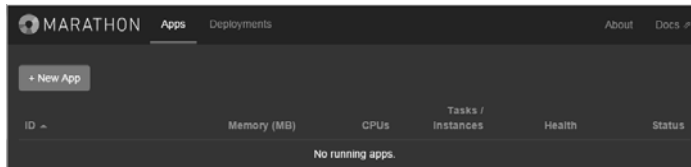


图 10-2 Marathon Web UI 界面

Marathon 同样采用 ZooKeeper 作为高可用中的 Leader 选举组件，其默认开启了高可用模式，如果需要关闭，则在启动参数中加入--ha，并设置为 false。在上例启动时我们通过--zk 接入 ZooKeeper 中。

与 Mesos 不一样的是，在访问 Marathon 各节点时其并不会重定向到主节点，而是以 Proxy 方式将请求发送到主节点。

### 10.2.2 运行Shell程序

下面写一行简短的 Shell 代码，它将模拟一个服务器接收外部请求并发出响应，在 while 语句循环中，这意味着它将像服务器一样长期运行。代码如下：

```
while [ true ] ; do echo -ne "HTTP/1.0 200 OK\r\nContent-Length: 12\r\n" | nc -l -p 1234 ; done
```

在一台服务器上执行，在另一个节点的 telnet 1234 端口将看到如下效果：

```
ubuntu@slave1:~/command# telnet 192.168.121.128 1234
Trying 192.168.121.128...
Connected to 192.168.121.128.
Escape character is '^]'.
HTTP/1.0 200 OK
Content-Length: 12
hello world
Connection closed by foreign host.
```

编辑一个 JSON 格式的文本，在 cmd 中包含了上述 Shell 代码，保存为 example1.json 文件：

```
{
  "id": "example-1",
  "cmd": "while [ true ] ; do echo -ne 'HTTP/1.0 200 OK\r\nContent-
Length: 12\r\n' | nc -l -p 1234 ; done",
  "cpus": 0.01,
  "mem": 1.0,
  "instances": 1
}
```

接着在命令行中运行下面的语句：

```
ubuntu@slave1:~$ curl -X POST http://192.168.121.129:8080/v2/apps -d @
example1.json -H "Content-type: application/json"
```

其将 JSON 文件发送到 Marathon 调度器中，请求启动一个实例，资源为 0.01 core cpu，1MB 内存，执行单行 Shell 命令。需要注意的是 cmd 将发到 Mesos 的命令行 executor 进行执行，其执行格式是 `/bin/sh -c ${cmd}`

打开 Marathon 控制台，其默认端口是 8080，参见 <http://marathon:8080/>。

如图 10-3 所示，可看到一个名为 example-1 的 App 正在运行。在 Marathon 中 App 是一个完整的概念，用 App 来描述一个长服务，其可以是一组在多个机器上运行的服务。

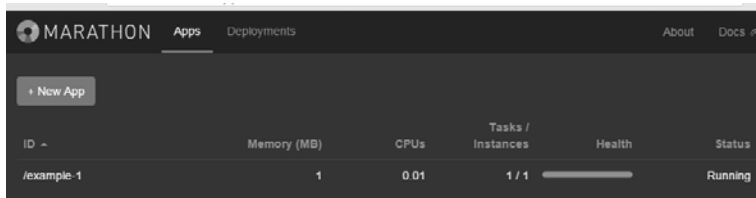


图 10-3 查看 Marathon Web UI 应用

接下来在 example-1 行可看到 App 的资源情况、健康状态及运行情况。单击 example-1，进入 App 内部的页面，如图 10-4 所示。

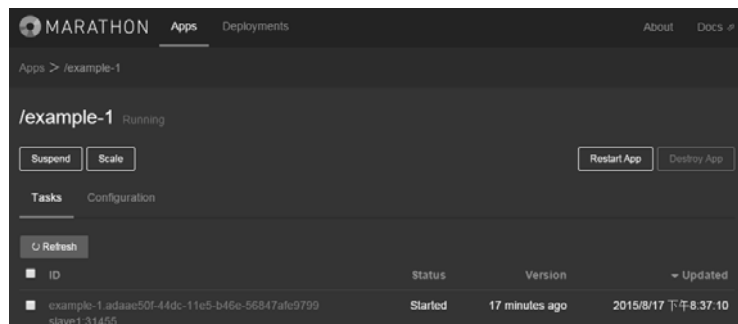


图 10-4 查看 Marathon Web UI 任务

在这里我们可以进一步对该 App 进行操作控制，选择 Scale 进行快速扩容，在弹出的菜单中选择 50，扩容到 50 个实例，如图 10-5 所示。

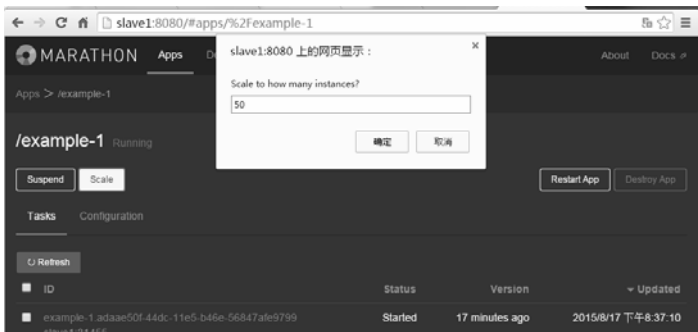


图 10-5 Marathon Web UI 应用扩容

很快会看到 50 个实例已经启动完成，如图 10-6 所示。

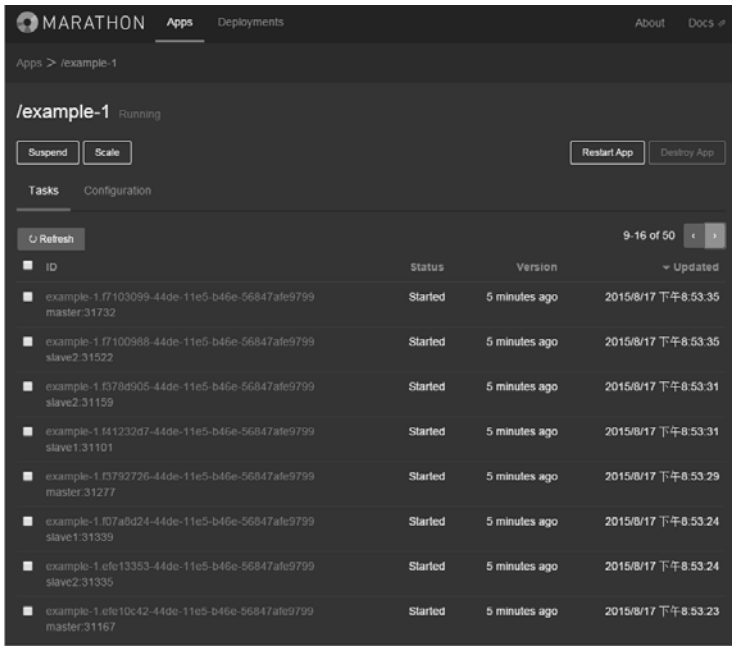


图 10-6 Marathon Web UI 扩容成功

Mesos 将命令行的执行放到了 Slave 节点的 Sandbox 中，等同于其有一个独立的运行环境，拥有自己的标准输入、输出。

使用 ps 命令查看在 Slave 节点执行的所有进程 ID:

```
ubuntu@master:~/command# ps -ef |grep while
ubuntu      5546   2760   1 05:56 ?          00:01:04 sh -c while [ true ];
do echo -ne 'HTTP/1.0 200 OK? Content-Length: 12? ' | nc -l -p 1234 ; done
ubuntu      6521   6485   0 05:55 ?          00:00:00 sh -c while [ true ];
do echo -ne 'HTTP/1.0 200 OK? Content-Length: 12? ' | nc -l -p 1234 ; done
ubuntu      6561   6525   1 05:55 ?          00:01:09 sh -c while [ true ];
do echo -ne 'HTTP/1.0 200 OK? Content-Length: 12? ' | nc -l -p 1234 ; done
```

```

... ..
ubuntu      14082 13910  1 05:55 ?          00:01:08 sh -c while [ true ];
do echo -ne 'HTTP/1.0 200 OK? Content-Length: 12? ' | nc -l -p 1234 ; done
ubuntu      17373 14289  1 05:56 ?          00:01:05 sh -c while [ true ];
do echo -ne 'HTTP/1.0 200 OK? Content-Length: 12? ' | nc -l -p 1234 ; done
ubuntu      20966 20598  1 05:55 ?          00:01:08 sh -c while [ true ];
do echo -ne 'HTTP/1.0 200 OK? Content-Length: 12? ' | nc -l -p 1234 ; done

```

使用 `lsuf` 命令查看其中一个进程打开的文件：

```

root@master:~/command# lsuf -p 27738
COMMAND  PID USER  FD   TYPE DEVICE SIZE/OFF  NODE NAME
... ..
sh        27738 root   0r    CHR   1,3      0t0    5861 /dev/null
sh        27738 root   1w    REG   8,1      216 157668 /var/lib/mesos/slaves/20150808-195125-2155456704-5050-2830-S1/frameworks/20150808-195125-2155456704-5050-2830-0000/executors/example-1.d6c60cae-44de-11e5-b46e-56847afe9799/ runs/9f3aa388-48e7-47f7-a57a-d74cbbe9abc6/stdout
sh        27738 root   2w    REG   8,1 1983751 157669 /var/lib/mesos/slaves/20150808-195125-2155456704-5050-2830-S1/frameworks/20150808-195125-2155456704-5050-2830-0000/executors/example-1.d6c60cae-44de-11e5-b46e-56847afe9799/runs/9f3aa388-48e7-47f7-a57a-d74cbbe9abc6/stderr
... ..

```

可以看到这个进程的标准输入并定向到了 `/dev/null` 之中，而标准输出及错误输出都定向到了 `Slave` 节点的任务相关文件夹中。

进入 `Web` 页面主界面，单击“+New App”，在弹出的选项框中填入如图 10-7 所示的内容。这次通过前台页面执行如下命令：

The image shows a 'New Application' dialog box with the following fields and values:

- ID:** example-2
- CPUs:** 0.1
- Memory (MB):** 16
- Disk Space (MB):** 0
- Instances:** 1
- Optional Settings:**
  - Command:** while [ true ]; do echo 'Hello Marathon'; sleep 5; done
  - Executor:** (empty field)

图 10-7 Marathon Web UI 新建应用

```
while [ true ] ; do echo 'Hello Marathon' ; sleep 5 ; done
```

### 10.2.3 运行远程资源

对于复杂的应用服务，我们没有办法将所有代码通过 `cmd` 传递到 `Slave` 节点中，通常情况下服务依赖于完整的计算逻辑，其可以是 `Shell` 脚本、`JVM` 命令等。但在执行之前需要保证的是计算逻辑在 `Slave` 的文件系统中已存在。`Marathon` 提供了 `URI` 的概念，在执行调度之前，利用 `Mesos fetcher` 功能下载、解压计算逻辑文件。

但在此之前让我们深入这个话题并看一个例子：

```
{
  "id" : " example-3 " ,
  "cmd" : " ./miniSrv.sh " ,
  "cpus" : 0.1 ,
  "mem" : 10.0 ,
  "instances" : 1 ,
  "uris" : [
    " https://example.com/app/miniSrv.sh "
  ]
}
```

上面这个例子将在执行 `cmd` 前通过 `Mesos` 下载资源，网址为 `https://example.com/app/cool-script.sh`，之后在 `Slave` 的 `sandbox` 中执行该资源脚本。我们可以查看 `Mesos` 的 `Web` 界面，查看该任务的 `sandbox`，单击进入界面，会发现 `Mesos` 下载的 `cool-script.sh` 脚本。`Mesos v0.22` 及以上版本在默认情况下执行 `cmd` 的方式是先设权限后执行，类似于 `chmod u+x miniSrv.sh && ./cool-script.sh`。

`Marathon` 除了直接执行资源文件，还可以对压缩文件进行解压，在遇到有以下后缀的文件时，`Marathon` 会尝试着进行解压：

- `.tgz`
- `.tar.gz`
- `.tbz2`
- `.tar.bz2`
- `.txz`
- `.tar.xz`

对于比较小的资源文件，我们可以将整个二进制文件打包放在本地服务器上，采用远程下载、解压运行的方式。其通过 `URI` 对资源进行定位下载，`Marathon` 支持多种下载源，包括如下几种。

- file:
- http:
- https:
- ftp:
- ftps:
- hdfs:
- s3:
- s3a:
- s3n:

uris 的值类型是数组，可以同时输入多个资源：

```
{
  ...
  "uris" : [
    "https://git.example.com/repo-app.zip", "
https://cdn.example.net/my-file.jpg", "https://cdn.example.net/my-other-
file.css"
  ]
  ...
}
```

#### 10.2.4 Artifact Store

PaaS 平台中有一个问题，即如何将可执行的计算逻辑打包发送到分布式的计算节点中，这势必在 PaaS 平台中有一个专门的计算逻辑存放仓库，它的具体形式可以是 VM 镜像仓库、Linux 包管理仓库、Docker 仓库，在 Marathon 内部已考虑到这种远程资源存放的需求，这个功能被称之为 Artifact Store。

在启动 Marathon 时，我们可以设置--artifact\_store 选项参数，配置其后端的存储系统，例如 hadoop 的 HDFS 分布式文件系统或者本地存储：

```
--artifact_store HDFS://hdfs_address:54310 /path/to/store
--artifact_store file:///yuhe/store
```

在前端，Marathon 提供了 REST API 接口，使用户可以上传、下载、更新、删除资源文件。

##### 1. 上传、下载文件

我们将之前的 Shell 代码编辑成文件，使用 curl 工具上传附件：

## PaaS 实现与运维管理：基于 Mesos + Docker + ELK 的实战指南

```
ubuntu@slave2:~$ curl -include -XPOST http://192.168.121.129:8080/v2/artifacts --form file=@/yuhe/miniSrv.sh
HTTP/1.1 100 Continue

HTTP/1.1 201 Created
X-Marathon-Leader: http://master:8080
Cache-Control: no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: 0
Location: http://master:8080/v2/artifacts/miniSrv.sh
Content-Length: 0
Server: Jetty(8.y.z-SNAPSHOT)
```

Marathon 将附件存储的 URL 地址在 Response 中返回。在上例中我们可以直接通过 `http://master:8080/v2/artifacts/miniSrv.sh` 来获取资源文件。

在 App 的 JSON 文档定义中，如果 `uris` 定义了多个资源文件，则在将任务下发到 Slave 后，每个实例都将下载一遍外部资源文件，这是一个非常消耗资源的过程。如果可以提前将这些资源文件上传到 Artifact Store 中，则从内部存储中获取所有资源应用可有效提升性能。

Marathon 提供了自动存储功能，在 App 定义文件中，我们将 `uris` 字段修改为 `storeUris`，格式如下：

```
{
  "cmd": "Python toggle.py $PORT0",
  "cpus": 0.2,
  "id": "app",
  "instances": 2,
  "mem": 32,
  "ports": [ 0 ],
  "storeUris": [ "http://downloads.mesosphere.com/misc/toggle.tgz" ]
}
```

以上述格式提交的请求，将按照下面的步骤处理：

- 在 Marathon 节点上下载资源文件；
- 将资源文件存储到本地 Artifact Store 中；
- 将资源文件在 Artifact Store 的 URI 地址添加到 App 定义文件中；
- 任务启动时从 Artifact Store 中下载资源文件。

登录到 Marathon 后台检查 App 定义文件，会发现其转换成了如下格式，`uris` 指向内部的 Artifact Store：

```
{
  "cmd": "Python toggle.py $PORT0",
  "cpus": 0.2,
  "id": "app",
  "instances": 2,
```



```

    "mem" : 32,
    "ports" : [ 0 ],
    "storeUrls" : [],
    "uris" : [ "
hdfs://localhost:54310/artifact/flcc046e4b603a94d0932eb818854fcc52e1b563/t
oggle.tgz" ]
  }

```

## 10.3 Docker容器运行

Marathon 与 Docker 的结合让应用可快速地运行到 IDC 不同的资源环境中。我们先以一个简单的示例开始。在下面的 JSON 格式中，使用了 Docker 的 Python:3 镜像，启动了一个容器内部端口为 8080 的服务，网络模式选择 bridge。在 portMappings 字段中 hostPort 被设置为 0，意味着将选择一个随机端口映射到外部。

```

{
  "id" : "example-3",
  "cmd" : "Python3 -m http.server 8080",
  "cpus" : 0.5,
  "mem" : 32.0,
  "container" : {
    "type" : "DOCKER",
    "docker" : {
      "image" : "Python:3",
      "network" : "BRIDGE",
      "portMappings" : [
        { "containerPort" : 8080, "hostPort" : 0 }
      ]
    }
  }
}

```

我们通过 HTTP API 接口启动这个应用：

```

ubuntu@slave1:$ curl -X POST http://10.141.141.10:8080/v2/apps -d
@example-3.json -H "Content-type: application/json"

```

接下来在 Marathon UI 中可以看到应用已经运行。

### 10.3.1 前提准备条件

本书使用的 Marathon 版本是 0.9.0，在启动 Docker 任务之前需要做好以下准备。

#### 1) 安装了 Mesos 及 Docker

所有计算节点安装了 Mesos slave，版本要求为 0.22.1+。

所有计算节点安装了 Docker，版本要求在 1.0.0 以上，本书使用的 Docker 版本为 1.7.1。

```
root@slave1:~/command# docker version
Client version: 1.7.1
Client API version: 1.19
Go version (client): go1.4.2
Git commit (client): 786b29d
OS/Arch (client): linux/amd64
Server version: 1.7.1
Server API version: 1.19
Go version (server): go1.4.2
Git commit (server): 786b29d
OS/Arch (server): linux/amd64
```

如果 Docker 版本较低，则可以在 Ubuntu 上使用如下命令直接进行升级：

```
sudo apt-get install lxc-Docker
```

### 2) 设置 Slave 容器类型

在 Slave 启动命令行中设置参数--containerizers，指定为 Docker、Mesos。

或者在配置文件中设置该参数：

```
ubuntu@slave1:$ echo 'Docker,mesos' > /etc/mesos-slave/containerizers
```

参数的设置顺序非常重要，它决定了在任务启动时选择的容器优先级。

### 3) 设置 Slave 的执行器超时时间

为了防止 Docker 下载镜像时间过长而导致任务超时失败，需要对 slave 的执行器超时时间进行设置。

在 slave 启动命令行中设置参数--executor\_registration\_timeout，指定为 5mins。或者在配置文件中设置该参数：

```
ubuntu@slave1:$ echo '5mins' > /etc/mesos-slave/executor_registration_timeout
```

第 2、3 步完成后需要重启 slave 以使之生效。

### 4) 设置 Marathon 任务超时时间

在 slave 中设置好超时时间之后，在 Marathon 上同样要设置好任务超时时间，可以在启动参数中设置--task\_launch\_timeout，至少保证该时间不短于执行器超时时间。其单位是毫秒，默认值为 300000，即 5 分钟。

## 10.3.2 端口资源分配

先不考虑容器中任务运行的情况，就一般任务而言，在分布式平台中由于并不知道任

务将运行在哪一台服务器上，所以如果任务所使用的监听端口相同，一旦两个任务运行到同一台服务器上，则会出现端口冲突，其中一个服务无法启动。例如我们启动两个 Tomcat 服务器，其监听端口都是 8080，两个任务被分配到一台主机上，后启动的任务将会失败。我们将这些任务内部的监听端口称为 **backend Port**。因为在一个 App 中会包含 1 个或多个实例，所以在前端会架设一个负载均衡器，它会提供一个唯一的入口地址供用户访问，其中也会有一个端口，在这里称为 **front Port**。

Marathon 的端口分配工作是围绕着 **front Port**、**backend Port** 进行的。对于 **backend Port** 的问题，Marathon 向 Mesos 申请随机端口解决。在任务中不需要指定专门的端口，而是读取 \$PORT 环境变量。对多个端口环境变量加一个编号，例如 \$PORT0，\$PORT1。

再回到之前的例子，将命令行写成如下内容：

```
{ echo -ne "HTTP/1.0 200 OK\r\nContent-Length: 12\r\n" ; echo "Hello
World" ; } | nc -l -p $PORT
```

保存的 JSON 文件格式如下：

```
{
  "id": "example-4",
  "cmd": "while [ true ] ; do echo -ne "HTTP/1.0 200 OK\r\nContent-
Length: 12\r\n" | nc -l -p $PORT ; done",
  "ports": [12345],
  "cpus": 0.1,
  "mem": 1.0,
  "instances": 1
}
```

在之前的例子中，netcat 命令的端口被指定为 1234，当任务运行到同一台机器时则会出现端口冲突报错，因此我们将其修改为 \$PORT，向 Mesos 申请随机端口，避免出现异常。

再来看看 Marathon 中的 **front Port**，这是一个并不存在的端口。**front Port** 由 Marathon 分配，在整个 Framework 内部是全局的、唯一的，它的主要用处在于服务发现。

可使用 `http://slave1:8080/v2/tasks` 查看该任务所分配的端口：

```
example-4 12345 slave2:31972
```

其中显示端口 12345 是 **front port**，而 31972 是随机分配的服务监听端口，即 **backend port**。Marathon 有自己的负载均衡脚本生成器，其依据以上信息生成内容。我们使用内置脚本生成配置文件，首先下载 marathon haproxy 配置工具：

```
wget https://raw.githubusercontent.com/mesosphere/marathon/master/bin/
haproxy-marathon-bridge
```

执行脚本，命令如下：

```
root@master:~/command/json# marathon-0.9.0/bin/haproxy-marathon-bridge
slave1:8080
```

它将输出 haproxy 的配置文件：

```
global
    daemon
    log 127.0.0.1 local0
    log 127.0.0.1 local1 notice
    maxconn 4096

defaults
    log                  global
    retries              3
    maxconn              2000
    timeout connect      5000
    timeout client       50000
    timeout server       50000

listen stats
    bind 127.0.0.1:9090
    balance
    mode http
    stats enable
    stats auth admin:admin

listen example-4-12345
    bind 0.0.0.0:12345
    mode tcp
    option tcplog
    balance leastconn
    server example-4-1 slave1: 31972 check
```

上面的 31972 是 backend ports，而 12345 是 front ports。backend ports 在 Marathon 中是随机分配的，而 front ports 可以随机分配，也能够指定。在上面的 JSON 文档中，我们使用 ports 字段对 front port 进行指定。

理解完 front port、backend port 后，我们再引入容器，容器与普通任务的区别在于其多了一个 container port。这是容器内部服务的监听端口，如果需要通过外部能够访问，则需要将端口映射到宿主服务器上，读者可再回顾 Docker 章节的内容。

整个端口的映射过程是 container port -> backend port -> front port。

Mesos slave 启动时的 --resources 设置会保留一批专用端口留给 Framework 使用，例如：

```
--resources=ports:[31000-32000];ephemeral_ports:[32768-57344]
```

Marathon 申请的 backend port 就是从上面的 31000~32000 中获取的，主动访问的随机端口则从 32768-57344 中获取。

### 10.3.3 容器端口分配

下面是一个 Docker 容器启动时的 JSON 格式文件：

```
{
  "id": "example-5",
  "cmd": "Python3 -m http.server 8080",
  "cpus": 0.5,
  "mem": 64.0,
  "instances": 2,
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "Python:3",
      "network": "BRIDGE",
      "portMappings": [
        { "containerPort": 8080, "hostPort": 0, "servicePort": 9000, "
protocol": "tcp" },
        { "containerPort": 161, "hostPort": 0, "protocol": "udp" }
      ]
    }
  },
  "healthChecks": [
    {
      "protocol": "HTTP",
      "portIndex": 0,
      "path": "/",
      "gracePeriodSeconds": 5,
      "intervalSeconds": 20,
      "maxConsecutiveFailures": 3
    }
  ]
}
```

我们着重看网络端口的分配，该容器启动的网络方式是 BRIDGE，在 portMappings 字段中有两组端口映射：containerPort 代表容器内部需要映射端口；hostPort 代表映射宿主主机的端口，最后 servicePort 代表这个 App 的服务端口，其分别表示 container port、backend port 和 front port。hostPort 字段为 0，因此它是随机分配的，我们也可以指定专属端口，这与非 Docker 任务不一样，但在 Slave 节点中端口出现重复时，同样会导致任务失败。

Marathon 提供简单的健康检查，例如 HTTP、TCP 等。在 JSON 文档中设置了对 Docker 任务的 HTTP 监控检查，频率（intervalSeconds）为 20s 每次，3 次失败（maxConsecutive Failures）则认为服务异常，每次失败后的间隔检查时间（gracePeriodSeconds）为 5s。

为了让容器像虚拟机一样使用，在 PaaS 平台中需要为每一个容器分配独立的 IP 地址，这个 IP 地址与宿主主机在同一网段。这样容器内部的所有端口可以无冲突地暴露到外部，

在本书写作时，Marathon、Docker 的官方网站并没有推出标准解决方案，这就需要客户化解决。可采用的方案如下。

- 修改 Docker 源码，在启动容器时动态获取 IP 地址，进行绑定。修改 Docker 源码的方式在架构上更加合理，调用关系也变得更简单，将同一功能聚合在 Docker 内部。但对于 Docker 源码的维护需要在企业内有专门的团队支持。
- 通过 pipework 外部修改容器 IP 地址。这种实现方式与 Docker 本身进行了分离，Marathon 有事件总线（Event Bus），在任务状态发生变化时会对注册其中的对象发送事件消息，这相当于一个回调函数，我们能够在网络模式为 none 的 Docker 启动后获得通知，之后远程命令通过 pipework 修改容器 IP 地址。

### 10.3.4 其他使用方法

#### 1. 强制下载镜像

Marathon 支持在启动任务之前强制 Docker 下载镜像文件，设置 forcePullImage 字段为 true 即可：

```
{
  "type": "DOCKER",
  "Docker": {
    "image": "group/image",
    "forcePullImage": true
  }
}
```

#### 2. Privileged 模式和任意参数

Marathon 支持 Docker 中的两个关键功能，即设置 Docker 的特权模式及在 Docker run 命令中传入任意参数。privileged 字段用来设置 Docker 的特权模式，其默认值为 false。parameters 字段是一个 key-value 的数组，用来让用户自定义传入 Docker run 命令行参数。这个字段最重要的作用在于 Marathon 的 Restful API 很难包含所有 Docker 命令行参数，因此用 parameters 作为动态参数输入选择。下面是一个示例文件：

```
{
  "id": "example-5",
  "container": {
    "Docker": {
      "image": "mysql",
      "privileged": true,
      "parameters": [
        { "key": "hostname", "value": "pingan.com" },
        { "key": "volumes-from", "value": "another-container" },
        { "key": "lxc-conf", "value": "... " }
      ]
    }
  }
}
```

```

    ]
  },
  "type": "DOCKER",
  "volumes": [],
},
"cpus": 0.2,
"mem": 32.0,
"instances": 1
}

```

## 10.4 Marathon管理

### 10.4.1 应用组

在 PaaS 平台中最小的计算单元是一个独立容器，由 1 个或多个同构容器实例构成一个集群。Marathon 将这样的集群称为应用（Application）。在实际生产环境中，一个应用由多种不同类型组件、不同业务模块组成，Marathon 将这种应用的聚合称为应用组（Application Group）。

业务系统的组成有两种常见场景：一种场景是一个业务系统由多种微模块单元构成，这些模块在不同的独立容器中运行；另一种场景是业务系统的栈中不仅包含了业务逻辑，还有缓存、数据持久化、负载均衡等相关组件。

应用组能很好地应对以上两种场景，用户可以将一个应用所包含的容器按照依赖做好定义。如图 10-8 所示是一个应用组的定义。

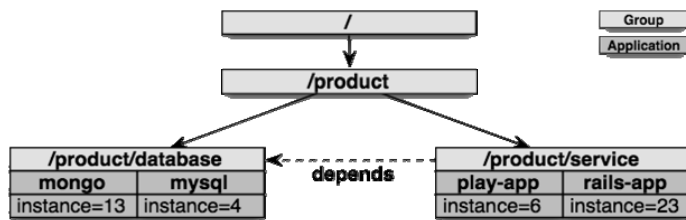


图 10-8 Marathon 应用组依赖关系

这是一个树型结构，根节点是组名称，可以包含其他子节点。子节点是子分组，包含了对应的应用。在图 10-8 中定义了一个 product 组，其下有 service、database 两个子节点组，其中一个组下有 play-app、rails-app 应用，另一个组下有 mongo、mysql 应用。

下面是对上述应用组定义的 JSON 格式文件：

```

{
  "id": " /product ",

```

```
"groups": [
  {
    "id": "/product/database",
    "apps": [
      { "id": "/product/mongo", ... },
      { "id": "/product/mysql", ... }
    ]
  }, {
    "id": "/product/service",
    "dependencies": [ "/product/database" ],
    "apps": [
      { "id": "/product/rails-app", ... },
      { "id": "/product/play-app", ... }
    ]
  }
]
```

## 1. 依赖关系

应用之间存在依赖关系，例如 `service` 应用依赖于数据库的运行，如果在 `Marathon` 的应用组定义文件中指定了这层关系，则其将跟踪应用的启动、停止过程来保证按照正确的次序执行。

依赖关系可以是应用级别的依赖，也可以是应用组级别的依赖。如果是应用组级别的依赖，那么组内的子分组、应用都会等待依赖对象服务可用后才会启动。依赖定义可以表述为相对路径，也可以表述为绝对路径。下面是对同一依赖项的定义：

```
{
  ...
  "dependencies": [ "/product/database" ],
  "dependencies": [ "../database" ],
  "dependencies": [ "specific/../../database" ],
  ...
}
```

## 2. 组内伸缩

应用可以通过调整 `instance` 字段进行伸缩。应用组同样有此需求，我们使用 `scaleBy` 字段进行设置。下面的配置将 `product` 组进行扩容，组内的应用将依据 `scaleBy` 字段扩容一倍。

```
PUT /v2/groups/product HTTP/1.1
Content-Length: 21
Host: localhost:8080
User-Agent: HTTPie/0.7.2
{ "scaleBy": 2 }
```



## 10.4.2 策略约束

在默认情况下，资源管理器在向 Framework 分配资源时，并不会设置一些约束策略，而是公平地分配资源，这就可能出现集群服务运行在一台硬件服务器上、关联组件运行在不同的机柜服务器中等现象。我们可以自定义一些策略，从而进一步保证服务的可靠性。

下面是一段约束策略示例：

```
"constraints": [[ "hostname", "CLUSTER", "a.specific.node.com" ]]
```

这个约束策略的含义是集群中的实例全部运行在一台服务器上。

约束策略由三部分组成：字段名称、操作符和一个可选的参数。下面对字段名称和操作等进行讲解。

### 1. 字段名称

字段名称如果是 `hostname`，则其对应 slave 节点的主机名称，字段名称也可以是 Mesos slave 节点的属性。在 slave 节点启动时，会设置 `--attributes` 选项，将当前资源的属性信息按照键值对应的方式列出，例如：

```
--attributes='rack:2;u:1'
```

上面的属性说明 slave 服务器的机柜位置为 2，u 位为 1。

在 Marathon 的字段名称中，可以定义 rack 或者 u。

### 2. 操作符

#### 1) UNIQUE

UNIQUE 是一种唯一性约束条件，它告诉 Marathon：“App 中的 Task 不能在共享资源中运行”。这些共享资源依据策略字段定义。例如，下面的策略只允许在一台服务器上运行 App 中的一个任务。

```
$ curl -X POST -H "Content-type: application/json" localhost:8080/v2/
apps -d '{
  "id": "unique",
  "cmd": "sleep 60",
  "instances": 3,
  "constraints": [[ "hostname", "UNIQUE" ]]
```

#### 2) CLUSTER

恰恰与 UNIQUE 相反，CLUSTER 让 App 的 Task 运行在共享资源中。共享资源依据策略字段进行定义。例如，下面的策略要求集群运行到 rack\_id 为 rack-1 的服务器中，这些服务器都在同一机柜中，从而降低了网络延时。

```
$ curl -X POST -H "Content-type: application/json" localhost:8080/v2/
```

```
apps -d '{
  "id": "cluster",
  "cmd": "sleep 60",
  "instances": 3,
  "constraints": [ "rack_id", "CLUSTER", "rack-1" ] ]
}'
```

### 3) GROUP\_BY

GROUP\_BY 让 Task 按某个字段属性均匀地分配到所有的资源上来保证高可用。例如下面的策略表示将 Task 按照机架位均匀发布，Marathon 将使用 rack\_id 字段的值来对资源进行分组。

```
$ curl -X POST -H "Content-type: application/json" localhost:8080/v2/
apps -d '{
  "id": "sleep-group-by",
  "cmd": "sleep 60",
  "instances": 3,
  "constraints": [ "rack_id", "GROUP_BY" ] ]
}'
```

如果我们并不想让 Task 分发到所有机架位中，则可以设置参数来限制分组数，例如传入参数，其值为 3，表示只在 3 个机架位中分发 Task，如下面的策略所示。

```
$ curl -X POST -H "Content-type: application/json" localhost:8080/v2/
apps -d '{
  "id": "sleep-group-by",
  "cmd": "sleep 60",
  "instances": 3,
  "constraints": [ "rack_id", "GROUP_BY", "3" ] ]
}'
```

### 4) LIKE

LIKE 接收一个正则表达式作为参数，允许我们将 Task 运行到字段值满足正则表达式的 slave 节点上。例如，下面的策略将 Task 发布到机架位 1~3 中。在 LIKE 操作符中，参数是必须添加的，否则会出现警告。

```
$ curl -X POST -H "Content-type: application/json" localhost:8080/v2/
apps -d '{
  "id": "sleep-group-by",
  "cmd": "sleep 60",
  "instances": 3,
  "constraints": [ "rack_id", "LIKE", "rack-[1-3]" ] ]
}'
```

### 5) UNLIKE

虽然和 LIKE 基本一样，但 UNLIKE 将 Task 运行到与正则表达式不匹配的 Slave 上。

```
$ curl -X POST -H "Content-type: application/json" localhost:8080/v2/
```

```
apps -d '{
  "id": "sleep-group-by",
  "cmd": "sleep 60",
  "instances": 3,
  "constraints": [[ "rack_id", "UNLIKE", "rack-[7-9]" ]]
```

### 10.4.3 健康检查

在 PaaS 内部实现健康检查功能有助于应用故障快速恢复，在 Marathon 内部就提供了这样的机制。监控管理与健康检查功能区域有重叠，但监控管理覆盖面比健康检查要宽泛很多，其包含了应用组成模块的各种指标，并汇聚成图形用于问题分析。而健康检查采用一种简单的方式来屏蔽或者恢复异常实例。

#### 1. healthy 状态机

Marathon 将应用的可恢复性与健康检查结合在一起，在状态发生变化时会触发 scaling 动作，保证可用服务实例数量，如图 10-9 所示是 Marathon 健康检查状态机示意图。

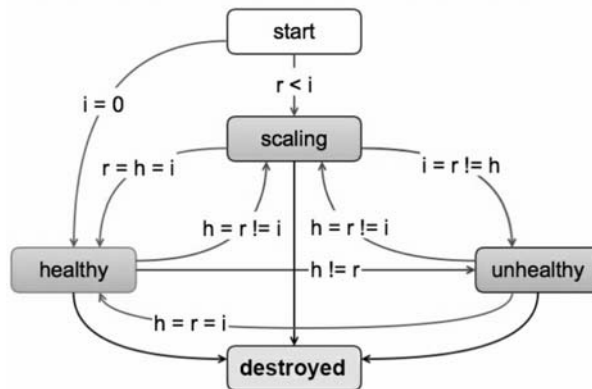


图 10-9 Marathon 健康检查状态机示意图

状态机共有三个活动状态：scaling、healthy、unhealthy，还有两个非活动状态：start、destroyed。

状态变化依据三个条件的算术计算逻辑进行组合，其分别是：请求实例数  $i$ ，运行实例数  $r$ ，健康实例数  $h$ 。

例如，当  $h=r \neq i$  时，即健康实例数等于运行实例数但不等于请求实例数时，状态将从 unhealthy 切换到 scaling，发起启动实例动作。

#### 2. 健康检查选项

Marathon 在健康检查中定义了相关选项，这些选项是探测服务可用的一般性逻辑。

### 1) intervalSeconds

每次健康检查的时间周期，默认为 60s。

### 2) gracePeriodSeconds

第一次服务状态变为 **healthy** 前，允许忽略检查失败的最长时间，默认为 300s。例如健康检查的周期是 60s，启动一个新服务后，前 4 次检查全部失败，由于当前总时长为 240s，还在默认的 300s 范围内，所以服务状态不会标记为 **unhealthy**，只有等到第 5 次检查结果出来后，才会判断是否更新服务状态。

### 3) MaxConsecutiveFailures

规定在多少次健康检查失败后将 **unhealthy** 服务，默认为 3s。如果设置为 0，那么在检查到失败后不做任务 **kill** 处理。

### 4) timeoutSeconds

健康检查等待的超时时间，超期后认为检查失败，默认为 20s。

### 5) protocol

为健康检查将采用的协议，Marathon 支持 TCP、HTTP、Command。Command 自定义命令行进行监控检查。要使 Command 有效，则必须在 Marathon 启动时设置 '--executor health\_checks' 选项，其表示在未明确指定执行器时的默认选择，默认为 "HTTP"。

### 6) path

健康检查的请求访问路径，该选择支持 HTTP，默认为 "/"，例如："/path/to/health"。

### 7) portIndex

在对服务进行监控时，发起的访问目的端口是 **backend port**，在 Marathon 中其实是随机分配的，并且在一个服务中可以存在多个端口，因此使用 **portIndex** 定义所监控端口的索引值，默认为 0。

### 8) ignoreHttp1xx

忽略 HTTP 返回状态码 100 to 199，默认为 false。如果返回状态码在这个范围内，则检查结果将被取消，任务的健康状态保持不变。

### 9) command

Marathon 的健康检查基于其最初的端口资源规则，对于 Docker 容器，服务端口及监听地址都会与这个规则不同，例如 Docker 容器要求像 VM 一样有自己的专属 IP 地址，并且每一个服务的端口都是专门指定的，所有 **instance** 都是一样的。这种情况下的监控检查必须采用 **command** 方式，使用外部命令行实现。

下面是健康检查的 JSON 文档示例。

采用 HTTP 进行健康检查：

```
{
  "path": "/api/health",
  "portIndex": 0,
  "protocol": "HTTP",
  "gracePeriodSeconds": 300,
  "intervalSeconds": 60,
  "timeoutSeconds": 20,
  "maxConsecutiveFailures": 3,
  "ignoreHttp1xx": false,
}
```

采用 TCP 进行健康检查：

```
{
  "portIndex": 0,
  "protocol": "TCP",
  "gracePeriodSeconds": 300,
  "intervalSeconds": 60,
  "timeoutSeconds": 20,
  "maxConsecutiveFailures": 0
}
```

采用 COMMAND 进行健康检查：

```
{
  "protocol": "COMMAND",
  "command": { "value": "curl -f -X GET http://$HOST:$PORT0/health" },
  "gracePeriodSeconds": 300,
  "intervalSeconds": 60,
  "timeoutSeconds": 20,
  "maxConsecutiveFailures": 3
}
```

#### 10.4.4 应用部署

在 Marathon 中应用或者应用组中的每一个改变都被称为部署。部署通常是一组动作：

- 启动或停止一个或多个应用；
- 升级一个或多个应用；
- 扩展一个或多个应用。

应用的部署是一个过程，需要花费一定的时间才能完成。在 Marathon 中从部署直到其完成前都处于互动状态。一个应用在同一时间只接收一个部署请求，在部署中的应用将拒绝其他部署请求。对于应用组部署而言，其中的应用将依据依赖关系完成部署动作。

### 1. 轮询重启

应用发布新版本是开发、运维人员都会面对的一个问题。通常发布管理会安排在非服务窗口进行，但随着互联网应用的需求变化、版本加速等，发布的次数、频率都产生了很大的变化，Marathon 关于应用发布、服务重启有一个 `minimumHealthCapacity` 选项，它是一个百分比参数，用于控制新实例的启动及旧实例的停止。

#### 1) `minimumHealthCapacity == 0`

在新实例启动前，将旧实例全部删除掉。

#### 2) `minimumHealthCapacity == 1`

在新实例启动后，再删除掉旧实例。

#### 3) `minimumHealthCapacity` 为 0~1

按照 `minimumHealthCapacity` 的比例，将旧实例缩容，之后按照相同比例启动新实例，等待健康检查全部通过后，停止所有旧实例，启动所有新实例。

例如，应用 A 有 10 个 instance，将 `minimumHealthCapacity` 设置为 0.6，整个发布过程如下：

- 停止 6 个旧 instance；
- 启动 6 个新 instance；
- 等待 6 个新 instance 健康检查通过；
- 停止剩的余 4 个旧 instance；
- 启动 4 个新 instance；

### 2. 强制部署

在一个时间点上，应用只接收一个部署请求，其他请求将放入队列中等待，直到当前部署完成。但这个规则可以通过设置强制部署标记打破。强制部署应当在部署失败的情况下使用。

如果强制标志被设置，那么被该部署影响的其他部署动作将全部取消，这可能会对应用造成不一致的状态。当一个应用在部署的升级重启中被打断时，它将保留在一个新、旧实例未完全启动、停止的状态。对于强制部署而言，唯一保证安全的场景是应用本身没有依赖关系。

使用强制部署时需要非常小心，只有需要纠正一个部署异常时才启用。

部署由一系列步骤构成，只有在上一步执行成功后才进入下一步。若存在以下情况则会导致整个部署无法完成：

- 一个新应用无法正确启动;
- 一个新应用无法检测正常;
- 依赖的新应用未申明或者无效;
- 集群的容量超过限制;
- ... ..

一旦出现以上情况, 则部署将进入迷失状态, 为了恢复应用、纠正错误, 必须采用一个强制部署。

### 3. REST API

Marathon 提供了关于部署的 REST API, 我们可以获取关于部署的相关信息。

- **affectedApps:** 哪些应用被该部署影响。
- **steps:** 该部署包含哪些步骤。
- **currentStep:** 当前执行的步骤。

这些步骤包括的动作有以下几种。

- **StartApplication:** 启动一个 **running** 的应用。
- **StopApplication:** 停止一个非 **running** 的应用。
- **ScaleApplication:** 改变应用实例的运行数。
- **RestartApplication:** 重启应用。
- **KillAllOldTasksOf:** 删除掉所有的旧实例。

下面是 JSON 文档示例。

#### 1) 请求

```
GET /v2/deployments HTTP/1.1
Accept: application/json
Accept-Encoding: gzip, deflate, compress
Content-Type: application/json; charset=utf-8
Host: localhost:8080
User-Agent: HTTPie/0.7.2
```

#### 2) 响应

```
HTTP/1.1 200 OK
Content-Type: application/json
Server: Jetty(8.1.11.v20130520)
Transfer-Encoding: chunked
```

[

```
{
  "affectedApplications": [
    "/test/service/srv1",
    "/test/db/mongo1",
    "/test/frontend/appl"
  ],
  "id": "2e72dbf1-2b2a-4204-b628-e8bd160945dd",
  "steps": [
    [
      {
        "action": "RestartApplication",
        "application": "/test/service/srv1"
      }
    ],
    [
      {
        "action": "RestartApplication",
        "application": "/test/db/mongo1"
      }
    ],
    [
      {
        "action": "RestartApplication",
        "application": "/test/frontend/appl"
      }
    ],
    [
      {
        "action": "KillAllOldTasksOf",
        "application": "/test/frontend/appl"
      }
    ]
  ],
  "version": "2014-07-09T11:14:11.477Z"
}
```

#### 4. 应用回滚

应用的每一次配置变化都会记录到版本中，在查看应用的所有历史版本时可以使用如下 API。

请求查看应用的所有历史版本信息：

```
GET /v2/apps/example-4/versions HTTP/1.1
Accept: application/json
Accept-Encoding: gzip, deflate, compress
Content-Type: application/json; charset=utf-8
Host: localhost:8080
User-Agent: HTTPie/0.7.2
```



响应结果列出了版本号：

```
HTTP/1.1 200 OK
Content-Type: application/json
Server: Jetty(8.y.z-SNAPSHOT)
Transfer-Encoding: chunked
```

```
{
  versions: [
    " 2015-08-22T10:41:51.564Z " ,
    " 2015-08-22T05:02:52.774Z "
  ]
}
```

提交请求回滚到指定版本：

```
PUT /v2/apps/example-4 HTTP/1.1
Accept: application/json
Accept-Encoding: gzip, deflate, compress
Content-Length: 39
Content-Type: application/json; charset=utf-8
Host: localhost:8080
User-Agent: HTTPie/0.7.2
```

```
{
  "version": " 2015-08-22T10:41:51.564Z "
}
```

### 10.4.5 事件总线

Marathon 在设计之初就考虑到了与外部接口程序的集成，除了被动地提供信息查询接口，还提供了主动的事件通知机制。在 Marathon 上订阅了事件通知的服务，可以接收关于应用、任务、部署等相关状态变化的通知。

#### 1. 订阅配置

我们可以在启动 Marathon 服务时设定订阅者，例如：

```
$ ./bin/start --master ... --event_subscriber http_callback --http_
endpoints http://host1/foo,http://host2/bar
```

host1 和 host2 将接收事件的通知。

使用 REST API 也可以进行事件订阅，示例如下。

事件订阅请求，callbackUrl 是接收事件服务的地址：

```
POST /v2/eventSubscriptions?callbackUrl=http://localhost:9292/callback
HTTP/1.1
```

```
Accept: application/json
Accept-Encoding: gzip, deflate, compress
Content-Length: 0
Content-Type: application/json; charset=utf-8
Host: localhost:8080
User-Agent: HTTPie/0.7.2
```

响应结果如下：

```
HTTP/1.1 200 OK
Content-Type: application/json
Server: Jetty(8.y.z-SNAPSHOT)
Transfer-Encoding: chunked

{
  "callbackUrl" : "http://localhost:9292/callback" ,
  "clientId" : " 0:0:0:0:0:0:0:1 " ,
  "eventType" : " subscribe_event "
}
```

## 2. 事件类型

Marathon 依据事件类型的不同发送通知，下面是 Marathon 发送的事件通知的 JSON 格式示例。

### 1) API 请求通知

在 Marathon 接收到关于应用变化（创建、更新、删除）的 API 请求时发送通知：

```
{
  "eventType" : " api_post_event " ,
  "timestamp" : " 2014-03-01T23:29:30.158Z " ,
  "clientId" : " 0:0:0:0:0:0:0:1 " ,
  "uri" : " /v2/apps/my-app " ,
  "appDefinition" : {
    "args" : [ ] ,
    "backoffFactor" : 1.15 ,
    "backoffSeconds" : 1 ,
    "cmd" : " sleep 30 " ,
    "constraints" : [ ] ,
    "container" : null ,
    "cpus" : 0.2 ,
    "dependencies" : [ ] ,
    "disk" : 0.0 ,
    "env" : { } ,
    "executor" : " " ,
    "healthChecks" : [ ] ,
    "id" : " /my-app " ,
    "instances" : 2 ,
    "mem" : 32.0 ,
    "ports" : [ 10001 ] ,
```

```

    "requirePorts" : false,
    "storeUrls" : [],
    "upgradeStrategy" : {
        "minimumHealthCapacity" : 1.0
    },
    "uris" : [],
    "user" : null,
    "version" : " 2014-09-09T05:57:50.866Z "
}
}

```

## 2) 任务状态通知

在任务状态发生变化时进行通知:

```

{
    "eventType" : " status_update_event ",
    "timestamp" : " 2014-03-01T23:29:30.158Z ",
    "slaveId" : " 20140909-054127-177048842-5050-1494-0 ",
    "taskId" : " my-app_0-1396592784349 ",
    "taskStatus" : " TASK_RUNNING ",
    "appId" : " /my-app ",
    "host" : " slave-1234.acme.org ",
    "ports" : [31372],
    "version" : " 2014-04-04T06:26:23.051Z "
}

```

在 Marathon 下任务有如下状态, 最后四种状态是终结状态:

```

TASK_STAGING
TASK_STARTING
TASK_RUNNING
TASK_FINISHED
TASK_FAILED
TASK_KILLED
TASK_LOST

```

## 3) Framework 消息通知

```

{
    "eventType" : " framework_message_event ",
    "timestamp" : " 2014-03-01T23:29:30.158Z ",
    "slaveId" : " 20140909-054127-177048842-5050-1494-0 ",
    "executorId" : " my-app.3f80d17a-37e6-11e4-b05e-56847afe9799 ",
    "message" : " aGVsbG8gd29ybGQh "
}

```

## 10.4.6 命令行参数

在 Marathon 启动时可以设置相关的命令行参数, 这些参数也可以通过环境变量传入,

其格式为 `MARATHON_OPTION_NAME`，即在原选项名称中加入前缀，例如 `MARATHON_MASTER` 为 `--master` 选项的环境变量名称。环境变量在命令行选项前读入，这意味着在两个地方同时设置了同一选项值，命令行参数将覆盖环境变量中的值。

### 1. 必填选项

Marathon 启动参数只有一个必填选项 `--master`，用于指定 Mesos Master 所在的位置。其格式是一个以逗号分隔的主机列表，例如 `zk://host1:port,host2:port/mesos`。如果 Mesos 没有使用 ZooKeeper 作为高可用组件，则直接填写 Master 所在的地址，例如 `http://localhost:5050`。

### 2. 可选选项

`--artifact_store`（可选默认值：无）：artifact store 的存储地址。例如：`" hdfs://localhost:54310/path/to/store "`，`" file:///var/log/store "`。

`--checkpoint`（可选默认值：`true`）：打开任务检查点。需要在检查点的 Mesos Slave 上启用。允许任务在 Mesos Slave 进程重启和 Marathon 调度器异常恢复后继续执行。

`--executor`（可选默认值：`"//CMD"`）：在没有指定时采用的默认执行器。

`--failover_timeout`（可选默认值：604800 秒，即 1 周）：在 Mesos 中以秒为单位。在 Marathon 与 Mesos 失去连接，或者 Marathon 异常退出后，只要在此时间内恢复，则其上的任务将继续运行，否则 Mesos Master 将删除所有任务。

`--framework_name`（可选默认值：`marathon`）：向 Mesos 注册的框架名。

`--ha`（可选默认值：`true`）：用于 Marathon 开启 HA 模式与 Leader 选举。允许在集群中运行多个 Marathon 实例，但需要在全局 HA 模式下启动，这种模式要求有一个运行的 ZooKeeper 作为协调者。

`--hostname`（可选默认值：机器主机名）：存储在 ZooKeeper 用户广播中的主机名，以便 Marathon 集群中的其他节点重定向到当选的 Leader。

`--Webui_url`（可选默认值：无）：Marathon Web UI 的 URL 地址，用来传递给 Mesos Master 以在 Mesos 页面中链接到 Marathon UI，默认设置成 Marathon Leader 主页地址。

`--leader_proxy_connection_timeout`（可选默认值：5000）：以毫秒为单位，表示 Marathon slave 实例与 Master 互连的超时时间。

`--leader_proxy_read_timeout`（可选默认值：10000）：以毫秒为单位，表示 slave Marathon 从 Leader 读取数据的超时时间。

`--local_port_max`（可选默认值：20000）：front 端口分配的最大端口号。如果应用程序定义静态分配的服务端口，则它不在这个范围内。

`--local_port_min`（可选默认值：10000）：front 端口分配的最小端口号。如果应用程序定

义静态分配的服务端口，则它不在这个范围内。

`--mesos_role`（可选默认值：无）：在 Framework 注册 Mesos 时使用。如果设置，则 Mesos 可以依据这个角色进行资源分配。

`--mesos_user`（可选默认值：当前用户）：在 Framework 注册 Mesos 时使用，标明执行任务的系统用户名。默认由以下条件决定 `SystemProperties.get`（“user.name”）。

`--task_launch_timeout`（可选默认值：300000 即 5 分钟）：以毫秒为单位，在删除任务前等待其进入 running 状态的最长时间。

`--http_endpoints`（可选默认值：无）：预配置的 HTTP 回调地址，也可以通过 REST API 动态设置。

`--zk`（可选默认值：无）：用来保存状态，实现高可用的 ZooKeeper URL 地址。格式为 `zk://host1:port1,host2:port2,.../path`。

`--zk_max_versions`（可选默认值：无）：每一个实体的版本数量限制。实体包括应用组、应用等。

`--zk_timeout`（可选默认值为 10000 即 10 秒）：连接 ZooKeeper 的超时时间。

`--zk_session_timeout`（可选默认值为 1.800.000 即 30 分钟）：ZooKeeper 会话超时时间，单位为毫秒。

`--mesos_authentication_principal`（可选）：用于 Mesos 身份验证的 principal。

`--mesos_authentication_secret_file`（可选）：用于 Mesos 认证密钥文件的路径

`--marathon_store_timeout`（可选默认值：2000，即 2 秒）：等待持久化存储完成的超时时间，以毫秒为单位。

## 10.5 服务发现

### 10.5.1 服务发现方法

服务发现用于解决应用之间相互通信的问题。“服务”是一个应用向外提供的门牌，这个门牌一般是一个具体的 IP 地址及服务监听端口。在传统企业应用中，门牌地址一般是固定的，变化频率很低。为了保证对关联方的影响最小，在 IP 地址之前加上了一层域名解析，因为有时 IP 地址会变化。“发现”其实是一个配置更新的问题，即如何将“域名+端口”门牌信息及时发送给关联方。我们先来讨论将这些信息注入到关联方的几种通用方法，对其优劣性进行讨论，之后介绍 Marathon 体系内的服务发现方法，以及 Mesos-DNS。

### 1. 配置文件

将服务配置信息放在文件中，在应用启动时加载读取。在传统企业应用中这是使用最普遍的方法，能够简单地集成现有代码，不需要启动额外的服务。但实时性方面必须要求服务采用域名方式。在需求更新快、版本多变的应用开发时代，每次进行环境切换后，例如从测试到 Stg，或者从 Stg 到 Prd，都需要更新配置文件。在 PaaS 平台中，由于计算单元都打包成了一个独立的包，因此每次切换都需要额外地做文件内容替换与同步，这就是配置文件的最大弊端。

### 2. key-value 库

用 key-value 库来存储配置信息是一种解决服务发现的新方法，例如 ZooKeeper 虽然以树形结构存储信息，但也可以看作以 key-value 方式存储信息。在实时性上，key-value 库会随着应用服务“门牌”的变化而变化，因此保持了足够的实时性。但其需要依赖于外部 key-value 服务，并且应用代码本身要支持这种注册、观察与接收通知的方式。这在兼容性上要复杂于文件方式。

### 3. 环境变量

在 12-factor 中，配置问题是通过环境变量解决的，服务发现是配置的一种。采用环境变量对应用的侵入小，可快速集成到代码中。另外它保持了在多套环境中切换的灵活性，让容器与变量分离，最后并不依赖于任何外部服务。

除了应用与应用之间的关联调用服务发现，另一层定义是面向于用户的服务发现。用户和应用的区别在于，应用是可以接收服务变化的通知而随之变化的；但对用户而言，其看到的一定是固定的“域名+端口”，如果这个入口变化了，则代表服务不可用了，或者服务已经变成了另外一个。

## 10.5.2 Marathon 方案

Marathon 的服务发现是针对其自身的 PaaS 规则构建的，这个规则的两个重要入口是我们之前讨论的 front Port 和 backend Port。

Marathon 会为应用中的任务分配 backend Port，这些任务会绑定到这个端口上来提供服务。这个端口的范围是 Mesos Slave 启动时进行资源设置设定的。Marathon 还定义了 front Port，这个端口本身是不存在的。在 Marathon 启动参数中有 --local\_port\_max、--local\_port\_min 两个选项，如果在应用 JSON 文档中不专门指定，那么 front Port 的分配将由小到大在这个范围内分配。front Port 并不真实存在，它扮演的角色是整个应用集群的入口端口。

例如，在一个 Tomcat Docker 容器集群中，在容器内部是 8080 端口，在 mapping 到外部后，如果不指定，则 Marathon 将分配一批 backend Port。这些 backend Port 可能相同，也

可能不同，但在外部应用访问这个集群时，只需要一个统一的端口，这个端口即 front Port。

Marathon 将 front Port 参数前缀定义为 local\_port，揭示了它的服务发现方案。它希望应用之间的访问可通过“localhost+front Port”来实现。

Marathon 首先在所有的 Slave 节点，或者说有 Marathon 任务的 Slave 节点部署 haproxy。之后在 Slave 上使用一个 Shell 脚本，haproxy-marathon-bridge 轮询地从 Marathon 中读取应用、任务信息，依据它们的端口信息生成配置文件，并动态加载到运行 haproxy 的进程中，默认轮询频率为 1 分钟 1 次。对于每一个任务而言，当它们需要访问应用时，会首先访问本机的“localhost+front Port”地址，haproxy 将接收这个请求，并转发到后端的真实应用中。

如图 10-10 所示是一个 Marathon 集群示意图，在集群中有两个应用：SVC1、SVC2，每个应用由两个任务构成。SVC1 与 SVC2 的 front Port 分别是 1111、2222，backend Port 是 31100、31200。

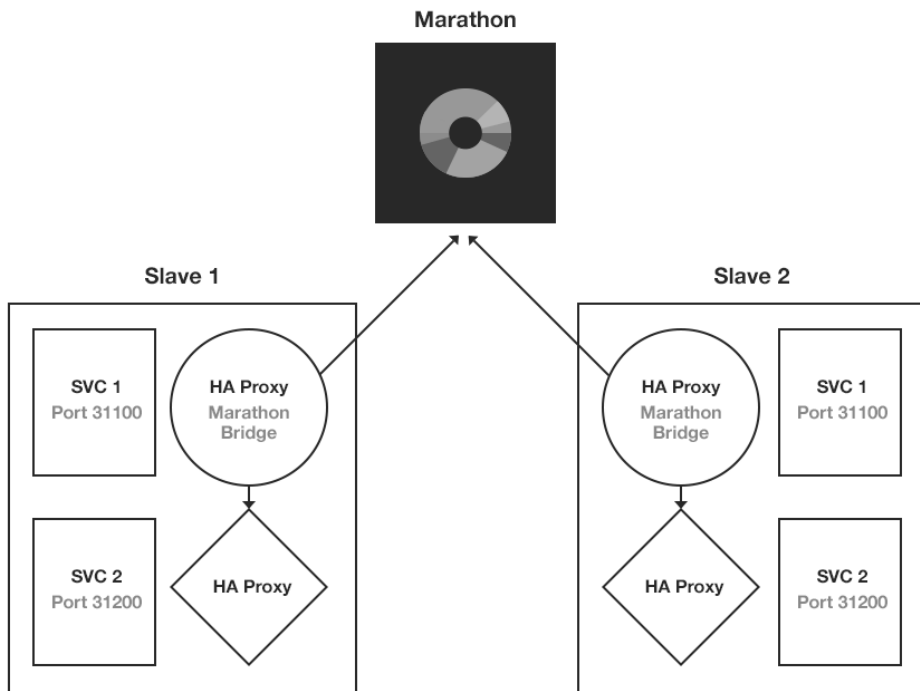


图 10-10 Marathon 集群示意图

SVC2 需要访问 SVC1 时，会访问本地服务器 haproxy 的 localhost:1111，haproxy 将路由这个请求到 SVC1 中的一个任务实例中。如图 10-11 所示，slave 2 上的 SVC 2 发送请求，被 haproxy 路由到了 slave 1 上的 SVC 1 中。

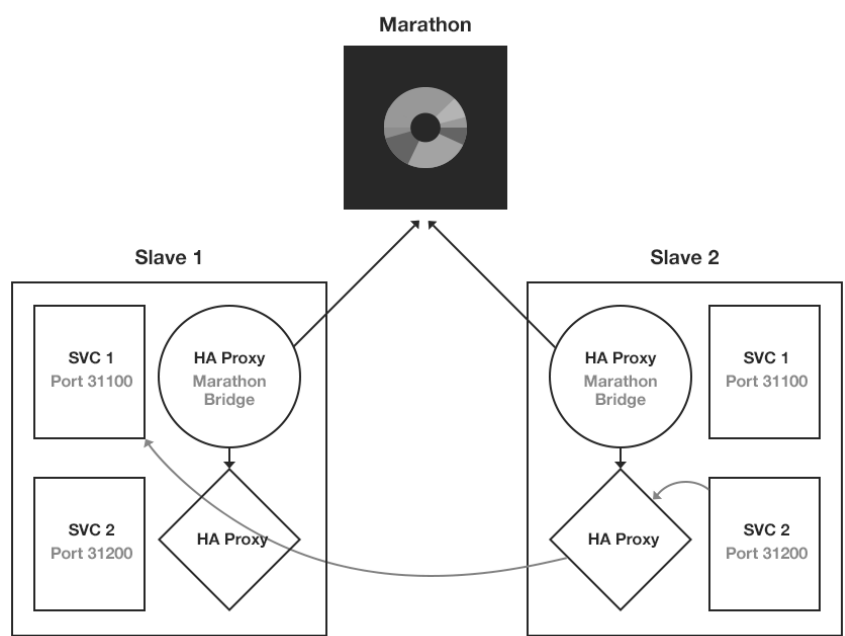


图 10-11 Marathon 集群服务发现

如果出现机器故障，slave 1 离线，则 SVC 2 的请求将通过 haproxy 转发到 slave 2 的 SVC 1 服务上，从而实现高可用。如图 10-12 所示。

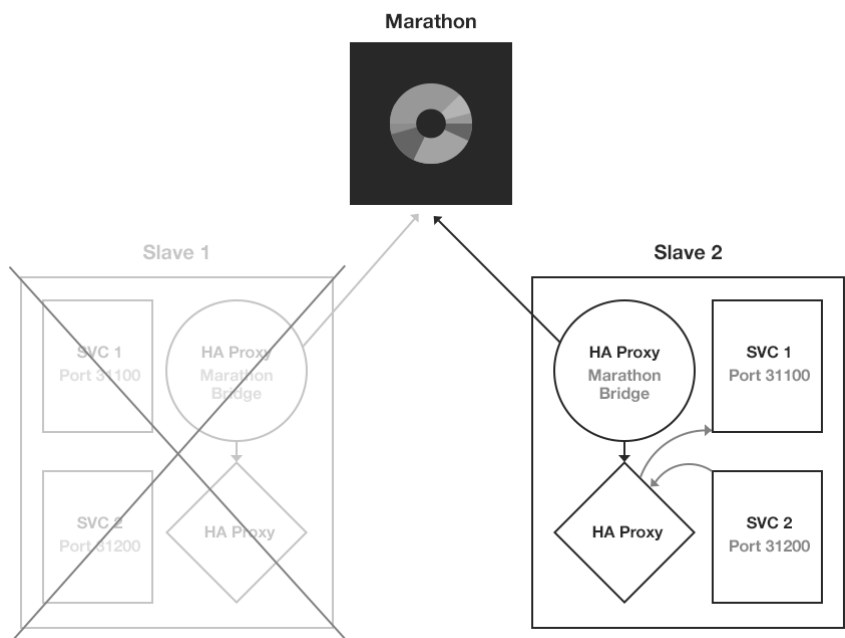


图 10-12 Marathon 服务高可用



要实现上述方案，则必须在所有计算节点安装 haproxy，轮询更新 haproxy 配置，对于大量的服务器场景，可以使用 salt、ansible 等软件配置工具的帮助。

为了安装 haproxy-marathon-bridge 脚本，需按照下述步骤操作。

(1) 下载 Shell 脚本，网址为 <https://raw.githubusercontent.com/mesosphere/marathon/master/bin/haproxy-marathon-bridge>。

(2) 创建 `/etc/haproxy-marathon-bridge/marathons` 文件，在文件中写入集群中所有 Marathon 服务的地址与端口，haproxy 脚本将自动从这里读取信息并访问 Marathon：

```
master.company.com:8080
slave1.company.com:8080
slave2.company.com:8080
```

(3) 执行脚本，将其部署成一个定时任务：

```
./haproxy-marathon-bridge install_cronjob
```

这个脚本将每隔一分钟更新一次 haproxy 的配置文件 `/etc/haproxy/haproxy.cfg`，如果它改变了，则 haproxy 会自动加载。

Marathon 提供的服务发现解决方案是在其内部应用之间使用的。对于外部应用调用的 Marathon 中的服务，在原有方案上，如果确保所有的 Slave 都有一个 Marathon haproxy 服务，那么在获知应用 front Port 的情况下，访问任意 Slave 节点地址即可访问该服务。

### 10.5.3 Mesos-DNS

在 Marathon 之外，Mesos-DNS 在 Mesos 集群中提供了另外一种服务发现方案。集群内的应用通过访问 DNS 服务器来获取对端的“门牌”信息。Mesos-DNS 是轻量级的无状态的 DNS 服务器，非常容易部署，如图 10-13 所示是其工作流程。

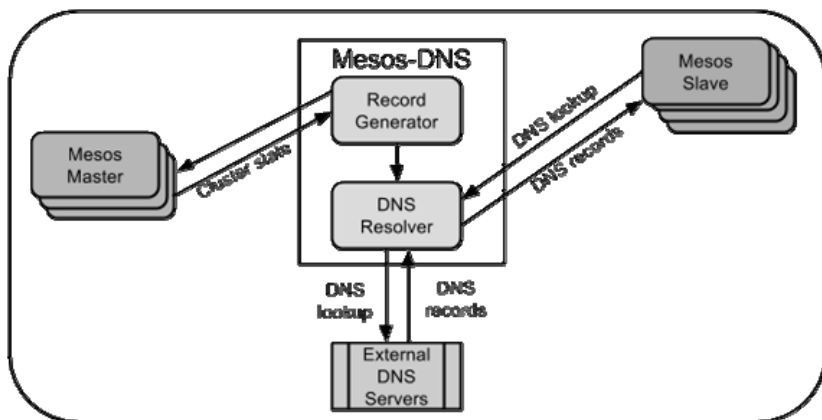


图 10-13 Mesos-DNS 工作流程示意图

Mesos-DNS 周期性地查询 Mesos Master，将所有 Framework 下的任务信息取回，之后为这些任务生成 DNS 记录（A 记录和 SRV 记录）。随着任务的启动、结束、失败等在集群中状态的变化，Mesos-DNS 将最新的状态信息更新到 DNS 中。Framework 并不需要与 Mesos-DNS 继续交互，在 Slave 中运行的应用将依赖于向 DNS 发起询问请求来获取关联方的 IP 地址、端口。

对于访问外部应用的情况，Mesos-DNS 中的所有域名可以设置在一个独立 Zone 中，之后将除这个 Zone 外的域名请求转发到外部 DNS 服务器上。

Mesos-DNS 是一个简单的无状态的服务，它不需要同步机制、持久化数据甚至进行文件备份，因此并没有实现心跳检测、监控、生命周期管理等功能。这些高可用、容错的功能可作为 Marathon 上的一个应用实现。Marathon 负责监控应用的健康状态，在服务失败之后重新启动，Mesos-DNS 能够从 Mesos Master 上找回最新的 DNS 记录状态。

### 1. 构建与运行

#### 1) 环境准备

在构建 Mesos-DNS 之前，需安装 Go 和 Godep。设置 GOPATH 环境变量指向到 Go 的安装目录中，之后设置 \$GOPATH/bin 到 PATH 环境变量中。

如果在安装 Go 时采用的自定义安装方式，那么需要设置 GOROOT 环境变量到相应位置，同时将 \$GOROOT/bin 添加到 PATH 环境变量中。下面示例展示了环境变量的设置：

```
export GOPATH=$HOME/go
export PATH=$PATH:$GOPATH/bin
export GOROOT=/usr/local/go      # 假设 Go 安装在 /usr/local/go
export PATH=$PATH:$GOROOT/bin
```

本书使用的 Go 原有版本是 1.4。

#### 2) 构建 DNS

使用 Godep 构建 Mesos-DNS：

```
go get github.com/mesosphere/mesos-dns
cd $GOPATH/src/github.com/mesosphere/mesos-dns
make all
```

执行完成后将生成一个 mesos-dns 文件，它是一个静态链接库，不再依赖其他库文件。目录下还有一个默认的配置文件的 config.json。

#### 3) 运行 DNS

我们将 mesos-dns 及配置文件复制到需要运行的主机，可以是 Mesos 集群中的任意机器。之后编辑配置文件 config.json，随后启动：

```
sudo mesos-dns -config=config.json &
```

采用 Marathon 来启动一个 Mesos-DNS 是一种不错的容错方式。如果 Mesos-DNS 出现异常，则 Marathon 将尝试重新启动它，确保服务不被中断。我们能够通过在 Marathon 中设置约束来保证 DNS 只运行在指定的服务器上。下面的示例显示，Marathon 从 JSON 文档中启动了一个 Mesos-DNS，并运行在主机 10.12.94.11 上。

```
{
  "cmd": "sudo /usr/local/mesos-dns/mesos-dns -
config=/usr/local/mesos-dns/config.json",
  "cpus": 1.0,
  "mem": 1024,
  "id": "mesos-dns",
  "instances": 1,
  "constraints": [[ "hostname", "CLUSTER", "10.12.94.11" ]]
}
```

#### 4) 设置 Slave

最后设置所有 Slave 节点上的/etc/resolv.conf 文件，让 Mesos-DNS 作为主 DNS 服务器使用。

## 2. 配置文件

Mesos-DNS 是通过一个 JSON 文档进行参数配置的，下面是配置参数示例：

```
{
  "zk": "zk://10.101.160.15:2181/mesos",
  "masters": [ "10.101.160.15:5050", "10.101.160.16:5050", "10.101.
160.17:5050" ],
  "refreshSeconds": 60,
  "ttl": 60,
  "domain": "mesos",
  "port": 53,
  "resolvers": [ "169.254.169.254" ],
  "timeout": 5,
  "httpon": true,
  "dsnon": true,
  "httpport": 8123,
  "externalon": true,
  "listener": "10.101.160.16",
  "SOAMname": "root.nsl.mesos",
  "SOARname": "nsl.mesos",
  "SOARrefresh": 60,
  "SOARetry": 600,
  "SOAExpire": 86400,
  "SOAMinttl": 60
}
```

zk 为 Mesos 集群上的 ZooKeeper 服务地址。其格式为 zk://host1:port1,host2:port2/mesos/，其中主机的数量可以是一个或多个。Mesos-DNS 将通过 ZooKeeper 找到当前 Mesos master

中的 Leader。

master 为以逗号分隔的 Mesos Master 列表，在 Mesos Master Leader 失效且无法进行响应时，Mesos-DNS 将按照此列表到所有 Master 节点上查询任务状态信息。zk 和 master 两个字段并不冲突，Mesos-dns 将先访问 zk，再查询 Master 列表，这两个参数都是静态的，因此在修改之后需要重启 Mesos-DNS。

下面是对配置文件为字段内容的说明。

- **refreshSeconds**: 从 Mesos Master 中检索任务信息，更新 DNS 记录的频率。默认值是 60 秒。
- **TTL**: 以秒为单位的 DNS 记录生存时间。它允许 DNS 记录在客户端缓存一段时间，以减少 DNS 的请求速度。ttl 的值应等于或大于 refreshSeconds。默认值是 60 秒。
- **domain**: 该 DNS 负责的 zone，默认值是 mesos。
- **port**: 该 DNS 的服务端口，默认值是 53。
- **resolvers**: 是一个逗号分隔的 IP 列表，它们是外部 DNS 服务器地址。当 DNS 请求的域名 zone 不在 Mesos-DNS 所负责的范围内时，将请求转发到这些外部 DNS 上。该字段是必填的。
- **timeout**: 请求外部 DNS 的超时时间。
- **listen**: Mesos-DNS 服务的监听地址，默认值是 “0.0.0.0”。
- **dnson**: 一个布尔字段，为用来控制 Mesos-DNS 是否启动 DNS 协议的请求服务。默认值是 true。
- **httpon**: 一个布尔字段，为用来控制 Mesos-DNS 是否启动 HTTP 的请求服务。默认值是 true。
- **httpport**: HTTP 的服务端口号，默认值是 8123。
- **externalon**: 一个布尔字段，用来控制 Mesos-DNS 是否提供 Mesos 自身 Zone 之外的请求服务。默认值是 true。
- **SOAMname**: Mesos Zone SOA 记录的 MNAME 字段。格式为 mailbox.domain，使用了 “.” 代替了 “@”。例如，如果电子邮件地址是 root@ns1.mesos，则该 MNAME 字段应该是 root.ns1.mesos。默认值是 root.ns1.mesos。
- **SOARefresh**: Mesos Zone SOA 记录的 Refresh 字段。默认值是 60。
- **SOARetry**: Mesos Zone SOA 记录的 RETRY 字段。默认值是 600。
- **SOAExpire**: Mesos Zone SOA 记录的 EXPIRE 字段。默认值是 86400。

- SOAMinttl: Mesos Zone SOA 记录的最小 TTL 字段。默认值是 60。
- recurseon: 控制 Mesos-DNS 是否允许递归查询。默认值是 true。

### 10.5.4 Bamboo

对于用户直接访问应用服务的服务发现需求，Marathon 的方案是满足其内部应用之间的互相访问的，端口会随时发生变化。而 Mesos-DNS 主要针对的是域名访问的方式，而且这些域名是与任务相关联的，对用户来说不具备可读性。

用户直接访问需求的要求有一个稳定的域名和端口地址。

这种需求必须自定义工作流。之前讨论过可以让容器使用独立的 IP 地址，为了让用户永远使用固定域名、端口访问服务，可以按照下面的步骤处理。

(1) 正常启动 Marathon Docker 应用任务。

(2) 任务启动完毕后，订阅者接收事件通知，启动一个固定服务端口的 haproxy 容器，在配置文件中注入后端任务服务地址。

(3) haproxy 启动完毕后，订阅者接收事件通知，对任务容器、haproxy 容器的 IP 地址进行域名注册。

如果不想全部自定义实现，则也有类似的开源组件可以使用。

Bamboo 是一个用来自动配置 haproxy 的服务。你可以认为它是一个 haproxy 容器，在调用启动之后，它将从 Mesos、Marathon 中读取相应的应用信息，用户可以通过 API 定义依据不同的 URL 或 hostname 来转发用户请求到后端应用。

Bamboo 服务发现机制如图 10-14 所示。

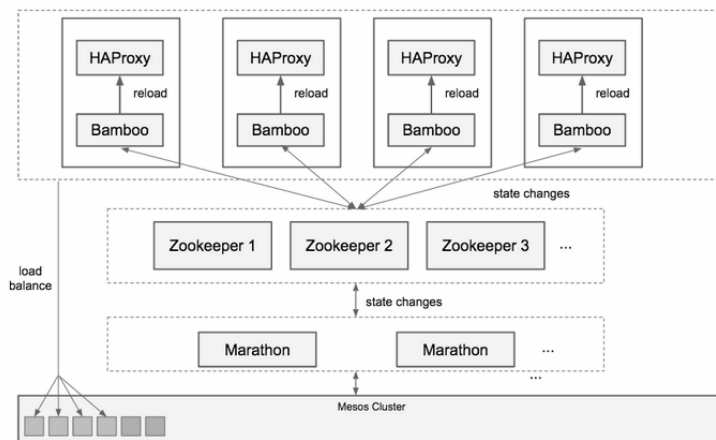


图 10-14 Bamboo 服务发现机制

Bamboo 可以作为容器独立运行，其中包含了负载均衡器 haproxy，以及一个获取应用信息、设置路由规则的 Bamboo 逻辑，下面让我们快速使用一个 Bamboo 服务。

下载 Bamboo 镜像并放入私有仓库：

```
Docker pull gregory90/bamboo:0.2.11
Docker tag gregory90/bamboo master:5000/bamboo
Docker push master:5000/bamboo
```

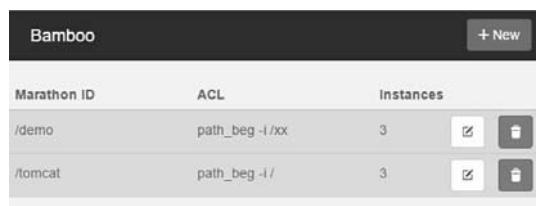
在任意 Slave 机器上 pull 镜像，并运行：

```
Docker pull master:5000/bamboo
Docker run -t -i -d -p 8000:8000 -p 80:80 \
    -e MARATHON_ENDPOINT=http://master:8080,http://slave1:8080,http://
slave1:8080 \
    -e BAMBOO_ENDPOINT=http://hostname:8000 \
    -e BAMBOO_ZK_HOST=master:2181,slave1:2181,slave2:2181 \
    -e CONFIG_PATH= " config/production.example.json " \
    -e BAMBOO_DOCKER_AUTO_HOST=true \
master:5000/bamboo
```

在启动容器时传入了相关的环境变量。

- **MARATHON\_ENDPOINT**: Marathon 服务的地址，用来订阅消息，获取信息。
- **BAMBOO\_ENDPOINT**: Marathon 订阅服务后的 callback 地址，Bamboo 上的 8000 端口专门用于订阅事件接收。
- **BAMBOO\_ZK\_HOST**: Bamboo 将把 proxy 信息存放到 ZooKeeper 中。
- **CONFIG\_PATH**: Bamboo 启动配置信息路径，采用默认值即可。
- **BAMBOO\_DOCKER\_AUTO\_HOST**: 对 Bamboo 容器的服务地址设定，默认设置为 true。

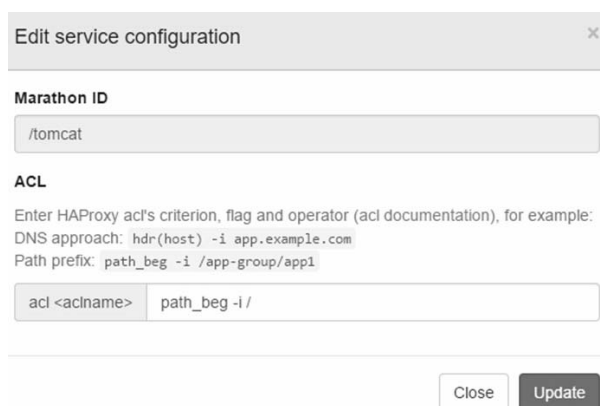
在 Bamboo 启动之后 80 端口默认用于提供应用服务，8080 端口用于后台管理及订阅消息接收。可通过地址 <http://ip:8000> 访问 Bamboo 服务，如图 10-15 所示，其检测到了 Marathon 中的两个应用：一个是/demo、一个是/tomcat。



Marathon ID	ACL	Instances		
/demo	path_beg -i /xx	3		
/tomcat	path_beg -i /	3		

图 10-15 通过 Bamboo 查看 Marathon 应用

Bamboo 支持 DNS hostname 匹配、URL Path 匹配两种模式，如图 8-16 所示为对应地用 Tomcat 进行设置，在 path 匹配到 “/” 路径后将请求转发到 Tomcat 应用。



Edit service configuration

Marathon ID

/tomcat

ACL

Enter HAProxy acl's criterion, flag and operator (acl documentation), for example:  
 DNS approach: hdr(host) -i app.example.com  
 Path prefix: path\_beg -i /app-group/app1

acl <aclname> path\_beg -i /

Close Update

图 10-16 Bamboo 设置匹配策略

在设置完成后，我们访问 Bamboo 的 80 端口，请求被转到了后端的 Tomcat 服务上。

## 10.6 Chronos 作业调度

### 10.6.1 作业调度框架

Marathon 在服务型应用调度中发挥着重要的作用。在 IT 日常任务中，除了面向用户的服务，还有大量的后台作业调度，这些作业包括数据备份、ETL、批处理等，它们在特定时间点被触发，定时执行任务，这些定时作业一般采用 Linux 的 cron 机制实现。这种方式难于维护，还非常容易出错。在容错性上，当作业所在节点出现异常时，作业是无法自动恢复的。

Chronos 是由 Airbnb 公司推出的用来替代 cron 的开源产品，它是一个具备容错特性的作业调度框架，可处理作业之间基于 ISO8601 的作业调度。我们可以用它来对作业进行编排，定义作业执行完成后的回调动作，支持任意长度的依赖链。Chronos 的内部脚本可以自动将文件传输到远程服务器上执行，同时支持在 Docker 容器中运行任务。

### 10.6.2 安装运行

下面让我们来安装并运行 Chronos。

(1) 安装 Mesos。

(2) 我们能够从 Github 中下载源码来构建 Chronos，在本书中将直接使用已编译好的压缩文件：

```
ubuntu@master: ~ $ wget http://downloads.mesosphero.io/chronos/chronos-2.1.0_mesos-0.14.0-rc4.tgz
```

(3) 解压，进入文件目录：

```
ubuntu@master: ~ $ tar xzf chronos-*.tgz
ubuntu@master: ~ $ cd chronos
```

(4) 设置 Mesos 动态链接库路径环境变量：

```
export MESOS_NATIVE_LIBRARY=/usr/local/lib/libmesos.so
```

(5) 使用 bash 脚本启动服务，它需要提供 Mesos Master 及 ZooKeeper 的地址，ZooKeeper 用于多个 chronos 节点的 Leader 选举，实现高可用。设置服务端口为 8081：

```
ubuntu@master: ~ $ ./bin/start-chronos.bash --master zk://192.168.121.128:2181/mesos --zk_hosts zk:// 192.168.121.128:2181/ --http_port 8081
```

(6) 启动 chronos 之后会自动向 Mesos 注册 Framework，打开 8081 Web UI 界面，如图 10-17 所示。

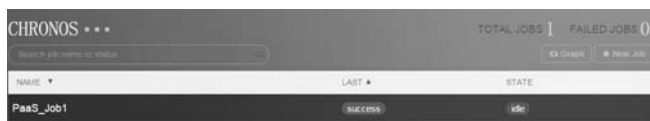


图 10-17 Chronos Web UI

### 10.6.3 作业示例

下面让我们创建一个 Chronos Web 作为示例进行演示（如图 10-18 所示），在 Chronos Web UI 上单击“+New Job”按钮，将弹出一个新建作业的详细信息输入框，在“COMMAND”中输入需要运行的命令，在示例程序中将运行一个简单的 sleep 程序。设定作业的执行时间，让它定时执行。如果我们不想按时间点来触发作业，则可以在 parent 字段中选择需要的其他作业，通过作业依赖触发。

作业一旦创建完毕，其在列表中的状态是“fresh”，一旦作业被调度执行，则它的状态将变为“success”或者“failure”。这些作业的执行在 Mesos 后台会以任务形式展现，从 Mesos Sandbox 中可以查看到作业的标准输出、错误输出等信息。

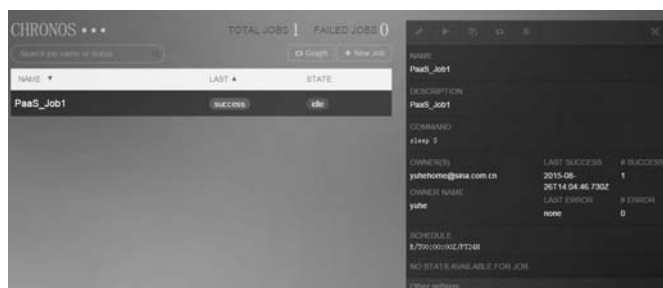


图 10-18 Chronos Web UI 新建作业



## 10.6.4 REST API

和 Marathon 一样，我们可以使用 Restful JSON API 与 Chronos 进行交互。Chronos 集群的 Leader 节点默认通过 8080 端口对外提供服务。在多节点的环境中，如果请求被发送到非 Leader 节点上，则它将被重定向到 Leader 上。

### 1. 新增定时作业

定时作业的新增方式是向 Chronos 服务端发出 Post JSON 文档的 HTTP 请求。文档结构是简单的 JSON 哈希键值对，需要包括以下字段。

- **Name:** 作业名称。
- **Command:** 实际需要执行的命令。
- **Schedule:** 作业的调度规则，遵循 ISO8601 规范。由三部分组成，通过 “/” 符号分割。例如 “R10/2012-10-01T05:52:00Z/PT2S” 的三部分内容如下。

(1) 重复执行作业的次数，如果仅有一个 R，则代表无限次执行。

(2) 开始启动作业的时间，如果为空，则表示立即执行，其遵循 ISO8601 规范，例如 YYYY-MM-DDThh:mm:ssTZD (eg 1997-07-16T19:20:30.45+01:00)

(3) 执行频率，其定义方式如下：

```
P10M=10 months
PT10M=10 minutes
P1Y12M12D=1 years plus 12 months plus 12 days
P12DT12M=12 days plus 12 minutes
P1Y2M3DT4H5M6S = P(eri)od 1Y(ear)2M(onth)3D(ay) T(ime) 4H(our)5M(inute)6S(econd)
```

其中，P 是必选字段，T 是可选字段，用来区分 M(inute)和 M(onth)。

- **ScheduleTimeZone:** 用来进行作业调度的时区。
- **Epsilon:** 指因某些原因 Chronos 丢失了运行作业的下一次时间时，采用的固定运行周期。
- **Owner:** 作业责任人的邮件地址。
- **Async:** 作业是否在后台运行。

下面是一个完整的作业 JSON hash 文档：

```
{ "schedule": "R10/2012-10-01T05:52:00Z/PT2S", "name": "SAMPLE_JOB1", "epsilon": "PT15M", "command": "echo 'FOO' >> /tmp/JOB1_OUT", "owner": "bob@airbnb.com", "async": false }
```

我们将上述文档提交到 chronos：

## PaaS 实现与运维管理：基于 Mesos + Docker + ELK 的实战指南

```
curl -L -H 'Content-Type: application/json' -X POST -d '{json hash}'  
chronos-node:8080/scheduler/iso8601
```

### 2. 添加依赖作业

添加依赖作业与定时作业的 JSON 文档格式基本一样，仅需要将 `schedule` 字段修改为 `parents` 字段，该字段用来描述在启动作业前依赖的父作业。

添加依赖作业的 URL 地址如下：

```
curl -L -X POST -H 'Content-Type: application/json' -d '{dependent  
hash}' chronos-node:8080/scheduler/dependency
```

### 3. 添加 Docker 作业

一个 Docker 作业的任务格式与定时作业、依赖作业基本一致，在配置上多出了部分参数用于描述 Docker container，其中包括类型、镜像、网络、卷，以及需要分配的相关资源。下面是一个 Docker 作业 JSON 格式的示例文档：

```
{  
  "schedule": "R\2014-09-25T17:22:00Z\PT2M",  
  "name": "Dockerjob",  
  "container": {  
    "type": "DOCKER",  
    "image": "libmesos/ubuntu",  
    "network": "BRIDGE",  
    "volumes": [{"containerPath": "/var/log/", "hostPath": "/logs/", "  
mode": "RW"}]  
  },  
  "cpus": "0.5",  
  "mem": "512",  
  "uris": [],  
  "command": "while sleep 10; do date =u %T; done"  
}
```

Docker 作业的提交 URL 地址与定时作业、依赖作业一致：

```
curl -L -H 'Content-Type: application/json' -X POST -d '{json hash}'  
chronos-node:8080/scheduler/iso8601
```

# 大数据调度框架 Spark

## 11.1 Apache Spark 介绍

为了提高资源利用率，IDC 数据中心将三大底层资源在不同的时间提供给不同计算类型的应用使用，例如在白天是面向服务型的用户请求应用，在夜间用户请求减少后，这批正在运行的资源可供批处理、大数据等应用共享使用。本章我们将介绍如何在 Mesos 上运行 Apache Spark，共享底层资源。

### 11.1.1 Apache Spark 是什么

Spark 是一个 Apache 项目，被标榜为“快如闪电的集群计算”，其拥有一个繁荣的开源社区，并且是目前最活跃的 Apache 项目之一。Spark 提供了一个更快、更通用的数据处理平台。和 Hadoop 相比，Spark 可以使你的程序在内存中运行时的速度提高 100 倍，或者在磁盘上运行时的速度提高 10 倍。在 100 TB Daytona GraySort 比赛中，Spark 战胜了 Hadoop，它只使用了十分之一的机器数量，但运行速度提高了 3 倍。Spark 也已经成为针对 PB 级别数据排序速度最快的开源引擎。

Spark 的核心概念是 RDD（Resilient Distributed Dataset），它指的是一个只读的、可分区的分布式数据集，这个数据集的全部或一部分可以缓存在内存中，在多次计算间重用。传统的 MapReduce 虽然具有自动容错、平衡负载和可拓展性的优点，但是其最大缺点是采用非循环式的数据流模型，使得在迭代计算时要进行大量的磁盘 I/O 操作。RDD 正是解决这一缺点的抽象方法，它是一种有容错机制的特殊集合，可分布在集群的节点上，以函数式操作集合的方式进行各种并行操作。

传统的并行计算模型无法有效地解决迭代计算（Iterative）和交互式计算（Interactive）的问题，Spark 就是为此而生的，这也是它的价值所在。在迭代计算方面，Spark 的主要思想是 RDD，它把所有计算的数据保存在分布式的内存中。迭代计算在通常情况下都是对一个数据集做反复的迭代计算，数据在内存中将大大提高 I/O 操作，效率这也是 Spark 涉及

的核心：内存计算。在交互式计算方面，因为 Spark 是用 Scala 语言实现的，Spark 与 Scala 紧密集成，运用 Scala 的解释器，使其像操作本地集合对象一样轻松操作分布式数据集。

### 11.1.2 Lambda架构

现代的大数据架构由不同组件构成，用以满足相关需求。Lambda 架构是一个结合批处理、流式计算二者的优势来处理海量数据的大数据架构。这种架构设计通过批处理来保证数据的全面性、准确性，同时通过流式计算保证实时性，显示数据在线视图。它在数据处理的延迟、吞吐量、和容错性之间保持平衡。Lambda 架构描述由三层构成：批处理层（Batch Layer）、速度层（Speed Layer），以及响应查询请求的服务层（Serving Layer）。如图 11-1 所示是 Lambda 的架构图。

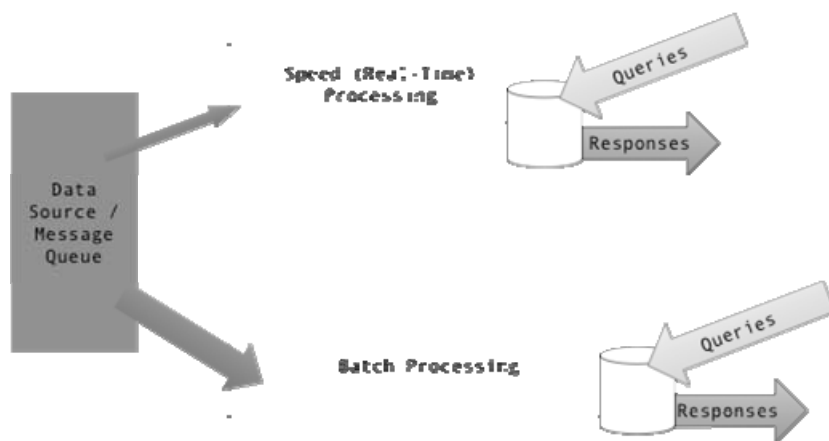


图 11-1 Lambda 架构图

- **Batch Layer:** 批处理数据（Batch Data Processing）通常意味着数据处理的完整性、精确性，在每次计算、修复数据结果时都是对整个数据集进行计算，结果将导出、覆盖到之前已计算的视图中，并以只读方式存在。批处理的时间跨度通常很长，一般为后台作业，用户异步等待数据处理完成通知。
- **Speed Layer:** 流式数据处理（Streaming Data Procession）则意味着保证数据处理的实时性，它以一种增量的方式处理、保证数据结果的低延时输出，并不强调数据的完整视图。流式数据处理的时间跨度很多，通常在数百毫秒到数秒之间。
- **Serving Layer:** 从批处理、流式计算中输出的结果将保存在服务层，这是一种预先准备好的数据视图，通过服务层交互式查询进行处理。

由于 Spark 的 RDD 具有丰富的表现力，转换与变化的灵活性使其成为 Spark 的核心，伯克利很快在此基础上衍生出一套同时满足以上三种数据架构情形的大数据平台。

### 11.1.3 Spark生态系统

Spark 生态系统由 Spark 核心及其一组功能强大的、高级别的库组成，这些库可以无缝地应用到同一个应用程序中。目前这些库包括 SparkSQL、Spark Streaming、MLlib，以及 GraphX。如图 11-12 所示是 Spark 生态系统简图。

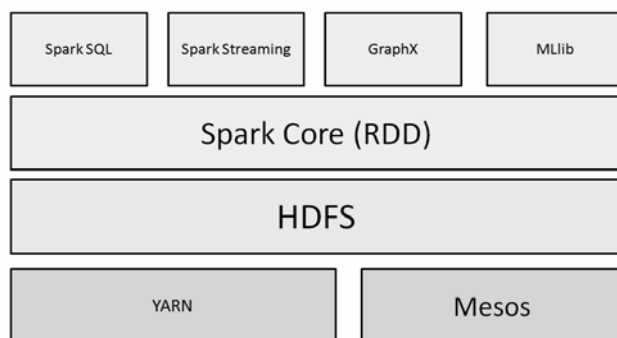


图 11-2 Spark 生态系统简图

#### 1. Spark Core

Spark Core 是一个基本引擎，用于大规模并行和分布式数据处理。它主要负责：

- 内存管理和故障恢复；
- 在集群上安排、分布和监控作业；
- 和存储系统进行交互。

这个基本引擎的功能实现是基于 RDD 的，RDD 可以包含任何类型的对象，它在加载外部数据集或者从驱动应用程序分发集合时创建。RDD 支持两种操作类型。

- 转换是映射、过滤、联接、联合等聚合操作，它在一个 RDD 上执行操作，然后创建一个新的 RDD 来保存结果。
- 行动是归并、计数、求最大值等计算操作，它在一个 RDD 上执行某种计算，然后将结果返回。

在 Spark 中，转换是“懒惰”的，即它们不会立刻计算出结果。相反，它们只是“记住”要执行的操作，以及要操作的数据集（例如文件）。只有当行为被调用时，转换才会真正地进行计算，并将结果返回给驱动器程序。这种设计让 Spark 运行得更有效率。例如，如果一个大文件要通过各种方式进行转换操作，并且文件被传递给第一个行为，那么 Spark 只会处理文件的第一行内容并将结果返回，而不会处理整个文件。在默认情况下，当我们在经过转换的 RDD 上运行一个行为时，这个 RDD 有可能会被重新计算。然而，你也可以通过使用持久化或者缓存的方法，将一个 RDD 持久化存储在内存中，这样，Spark 就会在集

群上保留这些元素，当我们下一次查询它时，查询速度会快很多。

### 2. SparkSQL

SparkSQL 是 Spark 的一个组件，它支持我们通过 SQL 或者 Hive 查询语言来查询数据。它最初来自于 Apache Hive 项目，运行在 Spark 上来替代太过底层的 MapReduce 接口，现在它已经被集成到 Spark 堆中。除了针对各种各样的数据源提供支持，它还使得代码转换与 SQL 查询编织在一起变得可能，这最终会形成一个非常强大的工具。

### 3. Spark Streaming

Spark Streaming 支持对流数据的实时处理，它将流式计算分解成一系列短小的批处理作业，利用 Spark 轻量级的调度框架处理数据，并根据批处理结果生成最终的流。Spark Streaming 会接收输入数据，它支持多种输入数据源，包括 Kafka、Flume、Twitter、TCP Sockets。

### 4. GraphX

GraphX 是一个库，用来处理图，执行基于图的并行操作。它为 ETL、探索性分析和迭代图计算提供了统一的工具。除了针对图处理的内置操作，GraphX 还提供了一个库用于通用的图算法，例如 PageRank。

### 5. MLlib

MLlib 是一个机器学习库，它提供了各种各样的算法，这些算法用来在集群上进行分类、回归、聚类、协同过滤等操作。其中一些算法也可以应用到流数据上，例如使用普通最小二乘法或者 K 均值聚类（还有更多）来计算线性回归。

## 11.2 Spark数据处理

### 11.2.1 Spark 运行模式

Spark 有两种运行模式：一种是本地模式，在本机运行计算；另一种则是分布式集群模式，利用分布式多节点完成计算任务。我们将焦点放在分布式集群模式上，基本组件与工作流程进行讨论，之后会介绍三种不同的分布式集群类型。

首先从开发人员的角度来看整个分布式集群模式下的所有组件。在 Spark 的运行起始点是开发人员写的应用程序。下面是一个简单的 Spark 应用 SimpleApp.java:

```
/* SimpleApp.java */
import org.apache.spark.api.java.*;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.function.Function;
```

```

public class SimpleApp {
    public static void main(String[] args) {
        String logFile = "YOUR_SPARK_HOME/README.md";
        SparkConf conf = new SparkConf().setAppName("Simple Application");
        JavaSparkContext sc = new JavaSparkContext(conf);
        JavaRDD<String> logData = sc.textFile(logFile).cache();

        long numAs = logData.filter(new Function<String, Boolean>() {
            public Boolean call(String s) { return s.contains("a"); }
        }).count();

        long numBs = logData.filter(new Function<String, Boolean>() {
            public Boolean call(String s) { return s.contains("b"); }
        }).count();

        System.out.println("Lines with a: " + numAs + ", lines with
b: " + numBs);
    }
}

```

这个应用程序对 README.md 文件中包含字符 ‘a’ 和 ‘b’ 的行进行计数。虽然只是一个简短的程序，却涉及 Spark 中非常重要的概念。

对于所有的 Spark 程序而言，进行任何业务逻辑的前提条件是在一个 Spark 上下文中。SparkContext 声明的是 Spark 运行的模式及模式类型，例如使用分布式集群模式，模式类型为 Mesos。在确定好上下文后才可以申请到运行资源来构建运行环境。在上面的代码段中，通过类 JavaSparkContext 创建了简单的本地模式上下文。

应用程序通过读取 README.md 文件创建了一个 RDD，随后执行了一个 Cache 控制操作，对 RDD 进行缓存，以便后续重复使用。紧接着在这个 RDD 上进行转换操作，过滤包含字符 ‘a’、‘b’ 等内容，最后通过 count 函数对包含字符的行进行计数并输出。

通过 maven 编译以上代码，打包成一个 JAR 文件，使用 spark-submit 即可运行这个应用程序。我们将应用程序称为 Driver Program，其中包含的 SparkContext 用来协调集群中的资源运行应用。SparkContext 能够连接到不同的集群管理器，包括 Standalone、Mesos、Yarn。

如图 11-3 所示是 Spark 部署基本架构图。

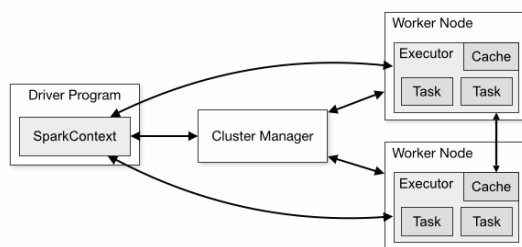


图 11-3 Spark 部署基本架构图

一旦连接上，Spark 便将在集群的计算节点上启动执行器 Executor，它们将为应用程序提供数据计算与存储。接下来，应用程序代码将被发送到这些 Executor 上，并且 SparkContext 分配任务 Task 到 executors 上运行。

在这个基本架构中有几点值得我们关注：

(1) Cluster Manager 是通用的资源分配器，它并不关注前面的是 Spark 应用还是其他资源申请者，例如 Marathon、Chronos 等。在 Cluster Manager 下所管理的资源可以在多种应用类型中共享，从而提升 IDC 资源使用率。

(2) 每个 Spark Driver Program 应用程序相当于一个 Framework，它将全面负责任务调度工作。每一个应用申请到的 Executor 进程运行着属于自己的应用逻辑与数据，在多个应用之间，即便它们的 Executor 在同一个主机上也无法共享数据。

(3) Driver Program 会直接与 Executor 进行交互，在整个应用生命周期过程中，它将监听一个服务地址，接收来自于 Executor 的连接。因此，Driver Program 与 Executor 必须保证网络可达，Driver Program 应当靠近 Worker Node 部署，最好在一个相同的局域网内。Driver Program 也有 RPC 远程调用开关来应对 Driver Program 与 Worker Node 分布部署的场景。

Spark 分布式集群模式有三种类型，分别是 Standalone、Spark on Mesos 和 Spark on YARN。其中，第一种类似于 MapReduce 1.0 所采用的模式，内部实现了容错性和资源管理；后两种则代表未来发展的趋势，部分容错性和资源管理交由统一的资源管理系统完成，让 Spark 运行在一个通用的资源管理系统之上，这样可以与其他 Framework 共用一个集群资源，降低运维成本和提高资源利用率，例如 Marathon。

Spark 可选择从源代码进行编译和安装，也可以直接下载 Pre-built 版本。在本书中我们使用了 Spark 1.4.1 for Hadoop2.6 的 Pre-built 版本。Spark 下载页面如图 11-4 所示。

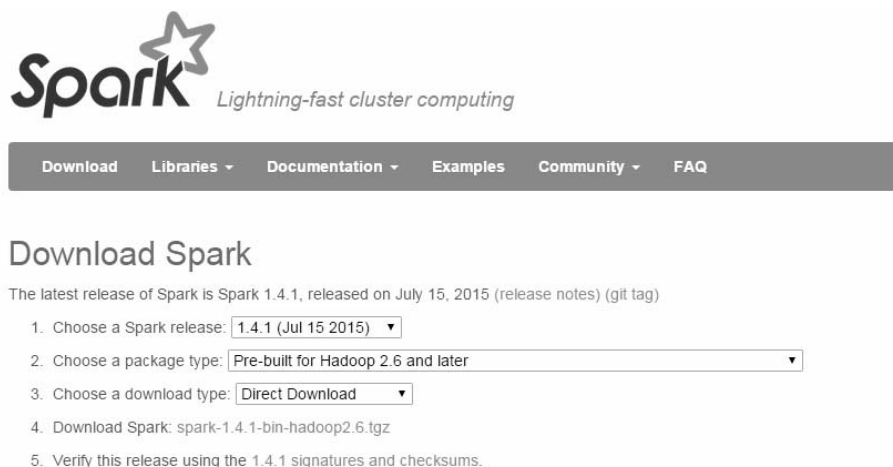


图 11-4 Spark 下载页面



(1) 在使用之前, Spark 要求已安装 Java 和 Git 到系统上, 我们先安装这两个依赖项, 在 Ubuntu 上使用下面的安装命令:

```
ubuntu@master:~$ sudo apt-get update
ubuntu@master:~$ sudo apt-get install openjdk-7-jdk git
```

(2) 解压下载的 Spark 压缩文件, 进入目录中:

```
ubuntu@master:~$ tar zxvf spark-1.4.1-bin-hadoop2.6.tgz
ubuntu@master:~$ cd spark-1.4.1-bin-hadoop2.6/
```

(3) 接下来启动 Spark Scala Shell, 它是一个修订版的 Scala Shell, 支持交互式数据分析与处理, 同样, 在 Spark Python Shell 中也可以用同样的方式启动。Spark-Shell 运作输出如图 11-5 所示。

```
ubuntu@master:~ $ ./bin/spark-shell
```

```
root@master:/yuhe/spark-1.4.1-bin-hadoop2.6# ./bin/spark-shell
log4j:WARN No appenders could be found for logger (org.apache.hadoop.metrics2.lib.MutableMetricsFactory).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
15/08/27 06:38:07 INFO SecurityManager: Changing view acls to: root
15/08/27 06:38:07 INFO SecurityManager: Changing modify acls to: root
15/08/27 06:38:07 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Set(root); users with modi
15/08/27 06:38:08 INFO HttpServer: Starting HTTP Server
15/08/27 06:38:08 INFO Utils: Successfully started service 'HTTP class server' on port 45866.
Welcome to

  ____ _
 / ___ \
/ /   \ \
/_ \   __ \
 \__ \  _/
  ___/  /
 /____/_/

 version 1.4.1

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_51)
Type in expressions to have them evaluated.
Type :help for more information.
15/08/27 06:38:28 INFO SparkContext: Running Spark version 1.4.1
15/08/27 06:38:28 INFO SecurityManager: Changing view acls to: root
15/08/27 06:38:28 INFO SecurityManager: Changing modify acls to: root
15/08/27 06:38:28 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Set(root); users with modi
15/08/27 06:38:30 INFO Slf4jLogger: Slf4jLogger started
15/08/27 06:38:30 INFO Remoting: Starting remoting
15/08/27 06:38:31 INFO Remoting: Remoting started; listening on addresses :[akka.tcp://sparkDriver@192.168.121.128:54958]
15/08/27 06:38:31 INFO Utils: Successfully started service 'sparkDriver' on port 54958.
15/08/27 06:38:31 INFO SparkEnv: Registering MapOutputTracker
15/08/27 06:38:31 INFO SparkEnv: Registering BlockManagerMaster
15/08/27 06:38:31 INFO DiskBlockManager: Created local directory at /tmp/spark-e4eala74-a3c7-4f38-b9b3-7ade37266d28/blockmgr-72de56c7-9ef5-48c7-977f-c1995a
15/08/27 06:38:31 INFO MemoryStore: MemoryStore started with capacity 267.3 MB
15/08/27 06:38:31 INFO HttpFileServer: HTTP File server directory is /tmp/spark-e4eala74-a3c7-4f38-b9b3-7ade37266d28/httpd-f1bd32b6-b7ac-4047-89c6-bc49c73b
15/08/27 06:38:31 INFO HttpServer: Starting HTTP Server
15/08/27 06:38:31 INFO Utils: Successfully started service 'HTTP file server' on port 49516.
15/08/27 06:38:31 INFO SparkEnv: Registering OutputCommitCoordinator
15/08/27 06:38:32 INFO Utils: Successfully started service 'SparkUI' on port 4040.
15/08/27 06:38:32 INFO SparkUI: Started SparkUI at http://192.168.121.128:4040
15/08/27 06:38:32 INFO Executor: Starting executor ID driver on host localhost
15/08/27 06:38:32 INFO Executor: Using REPL class URI: http://192.168.121.128:45866
15/08/27 06:38:33 INFO Utils: Successfully started service 'org.apache.spark.network.netty.NettyBlockTransferService' on port 34751.
15/08/27 06:38:33 INFO NettyBlockTransferService: Server created on 34751
15/08/27 06:38:33 INFO BlockManagerMaster: Trying to register BlockManager
15/08/27 06:38:33 INFO BlockManagerMasterEndpoint: Registering block manager localhost:34751 with 267.3 MB RAM, BlockManagerId(driver, localhost, 34751)
15/08/27 06:38:33 INFO BlockManagerMaster: Registered BlockManager
15/08/27 06:38:36 INFO SparkMLoop: Created spark context..
Spark context available as sc.
```

图 11-5 Spark-Shell 运行输出

(4) Spark 有一个 Web 界面可以展示应用程序上的相关信息, 可通过 `http://<driver-node>:4040` 访问。如果有多个应用程序运行在同一台机器上, 则端口被占用时, 它会自动扩展 (如 4041、4042 等)。Spark Web UI 界面如图 11-6 所示。

(5) Spark 在 `example` 目录中包含了很多 Spark 示例程序, 用于运行已经生成的 JAR 包中的代码, 如 Spark 自带的 Example 中的 SparkPi。

```
ubuntu@master:~ $ ./bin/run-example org.apache.spark.examples.SparkPi
local[3]
```

其中 local 代表本地，[3]表示有 3 个线程正在运行。

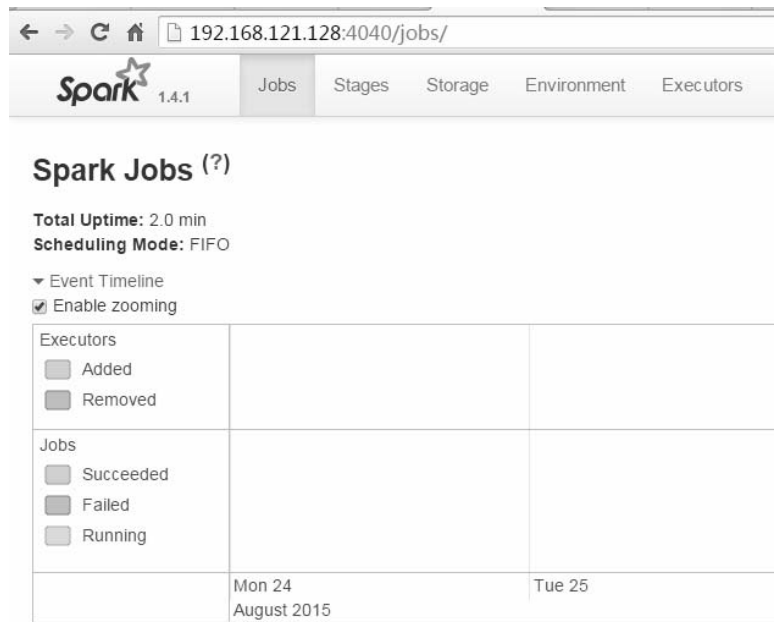


图 11-6 Spark Web UI 界面

### 11.2.2 Spark Standalone模式

Standalone 模式即独立模式，自带完整的服务，可单独部署到一个集群中，无须依赖任何其他资源管理系统。资源由 Master/Slaves 服务组成，借助 ZooKeeper 实现 Master 单点故障解决方案。各个节点上的资源被抽象成粗粒度的 Slot，与 Hadoop MapReduce 不同的是，Spark 不区分 Slot 类型（Map Slot 和 Reduce Slot），只有一种 Slot 可以供各种类型的 Task 使用，这种方式可以提高资源利用率，但失去了一定灵活性，不能为不同类型的 Task 定制 Slot 资源，Mesos 的资源分配模式很好地解决了这一问题。

#### 1. 快速安装启动

采用分布式集群方式，需要在所有计算节点上安装 Spark，保证部署目录一致。

##### 1) 在一台服务器上启动 Master

```
ubuntu@master:~ $ ./sbin/start-master.sh
```

服务启动之后，Master 会在两个服务端口上提供服务，其中一个业务接口，用来与 Slave 的 Worker 节点、应用程序的 SparkContext 通信。另一个是 Web UI，用来反映集群中的节点状态信息。在默认情况下 http://localhost:7077 为业务接口地址，http://localhost:8080

为 Web UI 接口地址，我们可以立即对其进行访问。

## 2) 在其他服务器上启动 Slave 节点

```
ubuntu@slave1:~ $ ./sbin/start-slave.sh <master-spark-URL>
ubuntu@slave2:~ $ ./sbin/start-slave.sh <master-spark-URL>
```

在 Slave Worker 节点启动之后，我们通过 Web UI 界面可以看到节点清单及其具备的资源情况。

以下参数可以在服务器启动时传递给 Master、Slave 节点，如表 11-1 所示。

表 11-1 Spark 启动配置参数

参 数	含 义
-h HOST, --host HOST	指定服务监听地址
-p PORT, --port PORT	指定服务监听端口 (Master: 7077, Worker 随机)
--Webui-port PORT	Web UI 端口 (master 默认: 8080, work 默认: 8081 for worker)
-c CORES, --cores CORES	允许 Spark 应用在此机器上可使用的最大 CPU 核数，只对 slave 节点有效
-m MEM, --memory MEM	允许 Spark 应用在此机器上可使用的最大内存数，只对 slave 节点有效
-d DIR, --work-dir DIR	用来存储临时文件、任务日志输出的目录，只对 slave 节点有效
--properties-file FILE	Spark 配置文件的路径 (默认: conf/spark-defaults.conf)

## 2. 集群启动脚本

手动逐台启动 Spark 服务非常低效，也很复杂，Spark 在 Standalone 模式下自带了一批脚本来方便我们批量运维 Spark 计算节点。首先在 Master 节点的 Spark Conf 目录下创建一个名为 slaves 的文件，该文件包含了集群中所有 Worker 节点的主机名或者 IP 地址，按行进行分隔。之后需要保证 Master 到其他 Worker 节点可采用 SSH 密钥登录，无须手动输入密码。

在 Master 主机上使用 SSH-keygen 生成公、私密钥对，执行如下命令后，按回车键。

```
ubuntu@master:~ $ SSH-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/rob/.SSH/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/rob/.SSH/id_rsa.
Your public key has been saved in /home/rob/.SSH/id_rsa.pub.
The key fingerprint is:
a6:5c:c3:eb:18:94:0b:06:a1:a6:29:58:fa:80:0a:bc rob@localhost
```

在 slave1、slave2 节点上创建.SSH 文件夹，将公钥内容输出到 authorized\_keys2 文件中。

```
ubuntu@master:~ $ cat ~/.SSH/id_rsa.pub | SSH ubuntu@slave1 "mkdir ~
/.SSH; cat >> ~/.SSH/authorized_keys2 "
ubuntu@master:~ $ cat ~/.SSH/id_rsa.pub | SSH ubuntu@slave2 "mkdir ~
/.SSH; cat >> ~/.SSH/authorized_keys2 "
```

在做好以上配置后，可以使用下面的脚本进行 Standalone 模式下的集群节点起停操作。

- `sbin/start-master.sh` 在脚本所在的机器上启动 Master 实例。
- `sbin/start-slaves.sh` 依据 `conf/slave.sh` 中的信息启动相关机器上的 Slave 实例，`sbin/start-slave.sh` 在脚本所在的机器上启动 Slave 实例。
- `sbin/start-all.sh` 依据 `conf/slave.sh` 中的信息启动机器上的 Master、Slave 实例。
- `sbin/stop-master.sh` 在脚本所在的机器上停止 Master 实例。
- `sbin/stop-slaves.sh` 依据 `conf/slave.sh` 中的信息停止相关机器上的 Slave 实例。
- `sbin/stop-all.sh` 依据 `conf/slave.sh` 中的信息停止机器上的 Master、Slave 实例。

以上这些脚本在 Master 所在的机器上运行。

## 3. 集群环境变量

通过 `conf/spark-env.sh` 配置文件，我们可以进一步对集群中运行的实例进行环境变量设置。在 `Conf` 目录下有一份参考文件 `spark-env.sh.template`，我们将它复制到 `spark-env.sh` 中，在此基础上进行修改，之后分发到所有节点上。如表 11-2 所示是主要的参考配置。

表 11-2 主要的环境变量参数

环境变量	含 义
<code>SPARK_MASTER_IP</code>	Master 节点所绑定的 IP 地址
<code>SPARK_MASTER_PORT</code>	Master 节点业务服务端口（默认为 7077）
<code>SPARK_MASTER_WEBUI_PORT</code>	Master Web UI 端口（默认为 8080）
<code>SPARK_MASTER_OPTS</code>	应用到 Master 节点上的配置属性，默认为空，其格式为 <code>form "-Dx=y"</code>
<code>SPARK_LOCAL_DIRS</code>	Spark 临时文件输出的目录位置，包括 Map 输出文件，RDD 磁盘存储文件等。这些目录应该放在访问速度快的本地存储系统中，它能够用逗号隔开来使用在不同磁盘上的多个目录文件
<code>SPARK_WORKER_CORES</code>	允许 Spark 使用的 CPU Core 数（默认可以使用所有 Core）
<code>SPARK_WORKER_MEMORY</code>	允许 Spark 使用的内存数
<code>SPARK_WORKER_PORT</code>	Spark Slave 节点启动后的业务服务端口（默认为随机）
<code>SPARK_WORKER_WEBUI_PORT</code>	Spark Slave 节点的 Web UI 端口（默认为 8081）
<code>SPARK_WORKER_INSTANCES</code>	用来设置每台 Slave 节点上可启动的 Worker 实例数，默认为 1，在一台资源充足的机器上可以尝试设置多个 Worker 实例。但我们可以设置 <code>SPARK_WORKER_CORES</code> 变量，保证每一个 Worker 所使用的 CPU 数，也可以不做设置，让 Worker 实例使用所有 CPU
<code>SPARK_WORKER_DIR</code>	用来存储临时文件、任务日志输出的目录，只对 Slave 节点有效（默认为 <code>SPARK_HOME/work</code> ）
<code>SPARK_WORKER_OPTS</code>	应用到 Slave 节点上的配置属性（默认为空）
<code>SPARK_DAEMON_MEMORY</code>	分配给的内存数（默认为 512m）
<code>SPARK_DAEMON_JAVA_OPTS</code>	进程的 JVM 选项参数，其格式为 <code>form "-Dx=y"</code>
<code>SPARK_PUBLIC_DNS</code>	Master、Slave 节点管理所使用的 DNS 服务器

### 11.2.3 Spark on Mesos

Spark On Mesos 模式是 Spark 最成熟的分布式集群模式。Spark 在设计之初就运行在 Mesos 上，Spark 的官方也推荐这种模式。Spark on Yarn 在 Spark 0.6 中被引用，但到了 branch-0.8 版本才真正可用。目前而言，Spark 运行在 Mesos 上会比运行在 YARN 上更加灵活、自然。在 Spark On Mesos 环境中，有两种调度模式可供用户选择来运行自己的应用程序。

#### 1) 粗粒度模式 (Coarse Grained Mode)

每个应用程序的运行环境由一个 Driver 和若干个 Executor 组成，每个 Executor 占用若干资源，内部可运行多个 Task。应用程序的各个任务在正式运行之前，向 Mesos 申请资源，一旦申请成功，则在整个运行过程中一直占用，即使 Task 不用，其他用户也无法使用，只有等到最后程序运行结束，这些资源才能被回收。例如，A 用户提交应用程序，申请使用 5 个 Executor 运行应用程序，每个 Executor 占用 5GB 内存和 5 个 CPU。Mesos 先分配资源，之后启动 Executor，Spark 开始进行任务调度。在应用程序的运行过程中，Mesos 的 Master 和 Slave 并不知道 Executor 内部各个 Task 的运行情况，Executor 直接将任务状态通过内部通信机制汇报给 Driver，我们可以认为，Spark 向 Mesos 申请了一组独立资源集群环境，与外部其他程序是隔离的。

#### 2) 细粒度模式 (Fine Grained Mode)

粗粒度模式的缺点非常明显，会造成一定的资源浪费，Spark On Mesos 还提供了另外一种调度模式：细粒度模式。细粒度模式类似于云计算的弹性扩容、按需分配，分配过程是随着运行情况的变化而动态进行的。与粗粒度模式一样，应用程序启动时，首先启动 executor，但每个 executor 所占用资源仅仅是运行所需的资源，不会对还未发起的任务进行资源分配。在运行过程中，Mesos 对每个 Executor 动态分配资源，每分配一些，便可以运行一个新任务，每个 Task 运行完成之后会立即释放资源。Task 会与 Mesos Master 和 Slave 进行通信，报告其运行状态，从而实现更加细粒度的资源管理和容错。细粒度模式的优点在于有效的资源控制，但缺点也很明显，频繁的消息通信、资源申请与释放将导致作业延时过高。

下面我们在 Mesos 上安装 Spark。

(1) 首先需要构建、安装 Mesos，可参见第 9 章。

(2) 下载 Spark Tar 压缩文件，本书下载的是 spark-1.4.1-bin-without-hadoop.tgz。

(3) 压缩包中含有 Mesos 需要使用的 Executor，因此需要将其放入分布式存储中共享，例如 HDFS：

```
ubuntu@master:~ $ hadoop fs -mkdir /spark
ubuntu@master:~ $ hadoop fs -put spark-1.4.1-bin-without-hadoop.tgz
/spark/ spark.tar.gz
```

(4) 通过 spark-env.sh.template 创建 spark-env.sh，添加三行相关环境变量：

```
ubuntu@master:~ $ cp spark-env.sh.template spark-env.sh
ubuntu@master:~ $ vim spark-env.sh
export MESOS_NATIVE_LIBRARY=/usr/local/lib/libmesos.so
export SPARK_EXECUTOR_URI=hdfs://master/spark/spark.tar.gz
export MASTER=mesos://zk://master:2181
```

MESOS\_NATIVE\_LIBRARY 指定动态链接库 libmesos.so 在所有节点上的位置，默认安装在/usr/local/lib/。

SPARK\_EXECUTOR\_URI 是 Spark 压缩文件在共享存储中的位置，在 Mesos 启动 Executor 时需要使用。

MASTER 用来指定 Mesos Master 所在的位置，可以指定到 Mesos 用于高可用的 ZooKeeper 服务上。

以上的环境变量参数是必须项，我们还可以设置其他参数，例如用 SPARK\_MASTER\_IP、SPARK\_WORKER\_PORT 指定 Spark Master 的监听地址等。

现在，Spark 在 Mesos 上已经部署完毕，接着可以创建 SparkContext，指定相关配置文件到 Mesos Master URI 及 Spark Executor 的位置，例如：

```
val conf = new SparkConf()
.setMaster( " mesos://zk://master:2181 " )
.set( " spark.executor.uri " , " hdfs://master/spark/spark.tar.gz " )
val sc = new SparkContext(conf)
```

在运行应用程序之后，我们在 Mesos Web UI 上可以看到一个新注册了的 Spark Frameworks，在任务页面可以看到执行情况，在 Sandbox 中可以查看每个任务的输入、输出。

在默认情况下 Spark 将运行在细粒度模式下，若对交互型应用有低延时需求，则需要进行配置，调整为粗粒度模式。模式由 spark.mesos.coarse 属性控制，通过在 SparkConf 中设置该属性为 true 来调整：

```
conf.set( " spark.mesos.coarse " , " true " )
```

除了 spark.mesos.coarse 属性，还有以下重要的配置参数可供调整，如表 11-3 所示

表 11-3 其他环境变量参数

属 性 值	默 认	含 义
spark.cores.max	none	指定 Spark 可以获取的最大 CPU 资源 Core 数，在默认情况下不做限制。
spark.executor.memory	none	指定 Spark 可以获取的最大内存数，在默认情况下不做限制
spark.mesos.extra.cores	0	该设置只对粗粒度模式有效，其设置额外的 CPU 核数。在这种模式下总的 CPU 资源 Core 数是所有任务申请的资源数加上额外的 CPU 核数，可以认为在粗粒度模式下不仅按当前任务数申请了 CPU 资源，还可以额外保留一部分作备用。需要注意的是 Executor 总的资源申请数不能超过 spark.cores.max

续表

属 性 值	默 认	含 义
spark.mesos.mesosExecutor.cores	1	该设置只对细粒度模式有效，分配给每一个 Executor 的 CPU 核数不包括用于任务执行的 CPU 核数。也就是说，即便没有任何任务运行，每一个 Executor 也会占用相应的 CPU 资源，该属性支持浮点数类型

同样，可以运行一个 Shell 命令，在 spark-env.sh 中定义好必选项，它将通过 SPARK\_EXECUTOR\_URI 找到 Spark Executor 文件的位置。在 spark-Shell 中传入 Mesos Master 地址或者其对应的 ZooKeeper 服务地址即可：

```
ubuntu@master:~ $ ./bin/spark-shell --master mesos://zk://master:2181
```

### 11.2.4 Spark Streaming

随着大数据的发展，人们对大数据的处理要求越来越高，批处理框架 MapReduce 适合离线计算，却无法满实时性要求较高的业务，则如实时推荐、用户行为分析等。Spark Streaming 是建立在 Spark 上的实时计算框架，通过它提供的丰富的 API、基于内存的高速执行引擎，用户可以结合流式、批处理和交互式查询应用。Spark Streaming 是 Spark 核心 API 的一种扩展，它实现了对实时流数据的高吞吐量、低容错率的流处理。数据可以有許多来源，例如 Kafka、Flume、Twitter、ZeroMQ、Kinesis 或 TCP 套接字，可以使用复杂算法对其处理实现高层次的功能，例如 Map、Reduce、Join。最后，经处理的数据可被输出到文件系统、数据库、和实时仪表盘。如图 11-6 所示。



图 11-6 Spark Streaming 输入、处理、输出

Spark Streaming 接收实时输入数据流并将数据划分批次，然后由 Spark Engine 分批处理，接着将生成的结果流输出到不同目的地。在第 12 章所讲解的日志处理框架中，也遵循了这样的一个处理流程：input→filter→output。Spark Streaming 可以用来进行日志处理与分析，它的强大之处是其借助后端的 Spark Core 分布式计算引擎，不仅提升了总处理效率，还提供了一种性能伸缩的解决方案。如图 11-7 所示。



图 11-7 Spark Streaming 内部工作原理图

## 1. DStream

Spark Streaming 提供了一个被称为离散流或 DStream 的高层次的抽象，它代表一个持续的流数据。DStreams 的创建可以是来自 Kafka、Flume 和 Kinesis 的输入数据流，也可以在其他 DStreams 上的应用的高级操作中创建。DStream 由连续 RDD 序列化构成。每个 RDD 含有一段时间间隔内的数据，如图 11-8 所示。



图 11-8 DStream 时间间隔窗口

任何在 DStream 上的操作都会转化为底层 RDD 操作，由 Spark Engine 处理，在后面示例中将使用的 flatMap 操作，如图 11-9 所示。

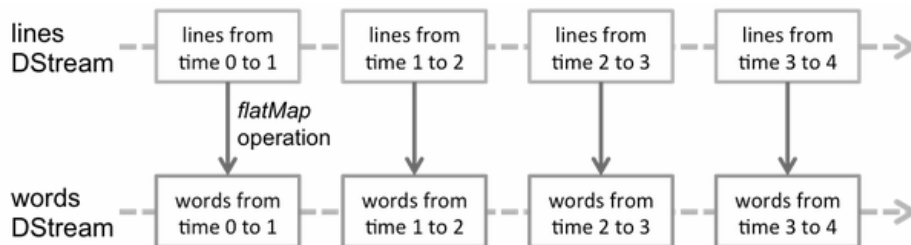


图 11-9 DStream RDD 操作

## 2. 简单示例

我们通过一个示例程序来介绍 Spark Streaming 的使用方法。首先导入 Spark Streaming 类名，随后建立一个 SparkConf，将其设置为 Mesos 集群模式，最后创建 StreamingContext，它是所有 Spark Streaming 功能的主入口点，定义数据收集的间隔时间为 1 秒。

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._

val conf = new SparkConf().setMaster("mesos://zk://master:2181").
setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))
```

通过这个 context 对象，可以创建一个新的 DStream 对象 lines，它作为客户端发起远程



访问，在这里是一个 TCP 的目标地址（localhost:9999）。lines 对象表示由远端服务器接收数据的数据流。这个 DStream 中的每个记录就是一行文字。接下来，我们要根据空格把每一行分割成单词。

```
val lines = ssc.socketTextStream("localhost", 9999)
```

flatMap 是一对多的 DStream 操作。它从源 Dstream 的每条记录中生成多个新记录到新创建的 DStream 中。在这里，每一行被分割为多个单词后保存到 words 的 DStream 中。接下来，我们将要计算这些单词的个数。

```
val words = lines.flatMap(_.split(" "))
```

Words DStream 通过 map 操作转换（一对一的转换）为一个 DStream(word, 1)键值对，然后通过 reduce 操作得到数据单词的频率。最后，wordCounts.print()打印每秒一批次的单词统计数。

```
import org.apache.spark.streaming.StreamingContext._
```

```
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)
wordCounts.print()
```

需要注意的是，这些代码行并没有立即执行，只有在设置了 Spark Streaming Context 计算启动后才会执行，启动调用如下：

```
ssc.start()
ssc.awaitTermination()
```

接下来让我们启动一个监听了 9999 端口的服务器，在这里使用 Netcat 工具：

```
ubuntu@master:~$ nc -lk 9999
```

以上示例程序包含在了 Spark 默认安装包中，它的应用名是 streaming.NetworkWordCount。在 spark-env.sh 中设置好环境变量，采用 Mesos 分布式集群模式：

```
ubuntu@master:~$ vim spark-env.sh
export MESOS_NATIVE_LIBRARY=/usr/local/lib/libmesos.so
export SPARK_EXECUTOR_URI=hdfs://master/spark/spark.tar.gz
export MASTER=mesos://zk://master:2181
```

运行 Spark streaming 程序：

```
ubuntu@master:~$ ./bin/run-example streaming.NetworkWordCount
localhost 9999
```

在启动 Netcat 的终端上输入一些单词：

```
ubuntu@master:~$ nc -lk 9999
hello world
... ..
```

接着，我们在 Spark Streaming 的终端可以看到，任何在 Netcat 中输入的行将被输出到

前端，每隔一秒输出一次，其输出如下：

```
ubuntu@master:~ $ ./bin/run-example streaming.NetworkWordCount localhost
9999
...
-----
Time: 1357008430000 ms
-----
(hello,1)
(world,1)
...
```

在示例程序中使用的是基于 TCP 源的数据流，为了集成其他数据源，例如 Kafka、Flume、Kinesis 等，我们需要引用其他相关 API 库，这部分并没有包含在 SparkCore API 中。这些数据源及其对应的 JAR 包如表 11-4 所示。

表 11-4 数据源及其对应的 JAR 包

Source	Artifact
Kafka	spark-streaming-kafka_2.10
Flume	spark-streaming-flume_2.10
Kinesis	spark-streaming-kinesis-asl_2.10
Twitter	spark-streaming-twitter_2.10
ZeroMQ	spark-streaming-zeromq_2.10
MQTT	spark-streaming-mqtt_2.10

# 日志集中管理 ELK

## 12.1 日志集中

### 12.1.1 日志集中介绍

在日志中包含了大量有价值的信息，它们无规律、非结构化地隐藏在分散的数据节点中。运维管理人员通过日志来诊断、分析问题；应用管理人员在应用日志中分析服务器响应效率及关键业务的变化。在安全方面，日志中会保留外部侵入的痕迹，以供问题分析。传统的日志分析方式是一个让人痛苦的过程，一般的步骤是先登录相关节点，打开日志文件查看，检索关键字，按照时间序列分析。日志价值的挖掘与个人系统管理能力相关，系统管理员会流畅地使用 `cat`、`less`、`grep`、`cut`、`awk` 等命令组合过滤文件，对于某些逻辑复杂、条件过多的日志分析，系统管理员会写出华丽的正则表达式脚本来分组、排序、过滤、拼装，从而完成日志分析任务。这种底层的日志分析方式的确可以培养一批运维脚本能手，但对于不熟悉运维或系统管理的其他人员来说，从日志中挖掘数据变得非常困难。

运维管理需要脚本能手、编程高手，但对于分布式环境下的问题诊断则变得不那么适用。分布式环境下的计算节点不再是一两个，而是成百上千个，应用的种类繁多复杂，分布在不同的操作系统、网络环境中；在 PaaS 平台上计算逻辑并不固定在一个节点上，它随时会因扩容、缩容等动作而在分布式环境中动态流动；日志不再是静态文件，而是一条条面向流的消息。华丽脚本的挖掘价值必须得到延伸。

应对分布式环境下日志分散的解决办法是收集日志，将其集中到一个地方。通过配置操作系统、网络设备，将日志发送到远端的中央管理服务器上。大量的开源工具适用于这样的场景，例如 `rsyslog`、`syslog-ng`。日志作为一条条消息传递到中央服务器上，集中存储。

收集好日志且集中存放，简化了运维人员在大量的机器节点的登录动作。紧接而来的是海量的日志，我们是否依旧用华丽脚本的方式对其进行处理？这种日志处理方式是否具备实时性，答案是否定的，将所有类型、所有节点的数据简单地堆叠在一起，日志量巨

大，在非结构化数据中进行查询处理会非常耗时，无法满足时间段检索或实时查询的要求。在海量数据中挖掘价值信息，形成有效的解决方案依旧艰难。

天下没有免费的午餐，如果想简单、有效地挖掘日志的价值信息，则在日志收集前就必须做大量的准备工作，将非结构化的数据变得半结构化，将半结构化的数据变得结构化。对日志最了解的人应当是日志管理的所有者，而不是将数据的处理放在一个团队，这样无法将日志的价值延伸与放大。运营人员关注日志访问，运维人员关注系统日志，网管人员关注设备日志，开发人员关注应用日志，我们没有办法要求一个人或者一个单位了解所有日志的内容、格式，并对其进行报表化。我们需要的是有一种足够开放、灵活的方法让所有关心日志的人在日志收集过程中对其进行定义、分割、过滤、索引、查询与报表定制。在很长一段时间内，日志集中只是在构建一套开放而灵活的系统，而真正发挥日志作用的是日志管理。

日志管理平台应当具备的能力包括：

- 1) 日志得到集中，汇聚在一起；
- 2) 灵活的日志格式化、过滤方法，开放给所需之人使用；
- 3) 索引日志内容，快速返回查询结果；
- 4) 具有伸缩性，在各个环节都能够扩容；
- 5) 具有高可用性，在单一节点失效的情况下不影响使用；
- 6) 强大的图形查询工具、报表产出工具。

### 12.1.2 日志集中架构

在构建日志集中管理的架构时，让我们回到问题的原点，将动作抽象化。日志到底是什么？它最可能的形式是一个文本文件或者一条网络消息。日志在什么地方？它可能在 Linux Windows 服务器上、网络设备或者任意其他位置。我们给这两个问题一个统称：输入，即 **input**，即我们从哪里获取日志。

**Input** 的下一步是对其进行处理，这个处理包括格式化、过滤、丢弃、替换等各类动作以让其满足要求，最简单的处理是什么都不做直接进入下一步，最复杂的处理是将多个处理动作揉和在一起，形成一个处理链，我们将这个处理统称为过滤即 **filter**。

过滤完成且格式化一条日志后进入最后一步，将这条加工完毕的日志，即输出 **output**。一个输出可能是将日志流存储到搜索引擎中，一个进程的输出是另一个进程的输入，日志流进入后续处理环节。

**input -> filter -> output** 是日志集中管理的最简范式。

日志集中管理的架构如图 12-1 所示。

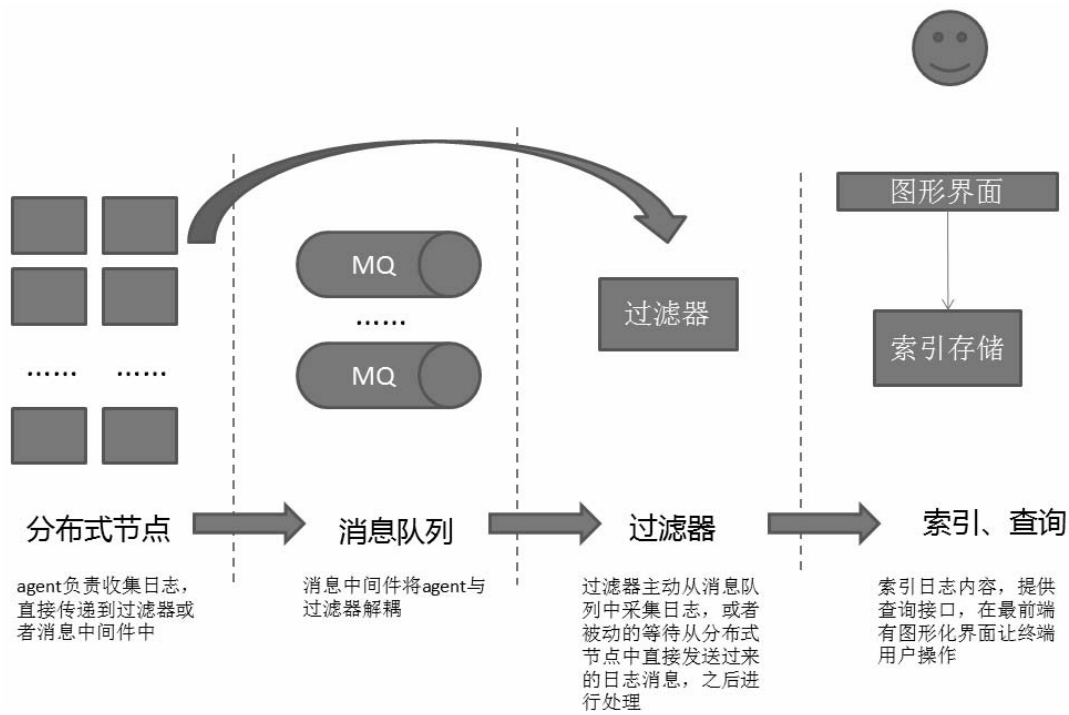


图 12-1 日志集中管理架构

在最简范式下将日志处理过程展开，在分布式节点的日志源端（服务器、网络设备、应用中间件）部署 agent，将日志发送到集中点。agent 的形态是开放型，可以是 syslog、logstash 等各类工具，在 agent 内部也遵循一个 input-filter-output 的范式。agent 本身也可以对日志进行处理，将过滤任务分发到各节点内部，对降低集中节点资源消耗有很大帮助。

agent 的日志输出有两条路径：一条是直接发送给日志过滤器，过滤器按照规则对日志进行格式化；另一条是与过滤器进行解耦，异步地将日志消息传递到中介消息队列中，等待过滤器获取日志。

agent 与过滤器直接通信存在一定的弊端，由于过滤器承担了消息处理的职能，在消息量过大的情况下，消息处理不及时很可能造成发送过来的日志丢失。另外，由于 agent 与过滤器之间的同步关系，agent 每次发送消息都需要等待过滤器返回，当过滤器缓慢时会影响到 anget 端的收集。在架构中引入 MQ 消息队列就是为了避免这两个问题。agent 的种类很多，在它的 output 模块允许的情况下，我们将消息优先放入消息队列中，过滤器主动从消息队列中获取消息。

过滤器承担着主要的日志格式化工作，处理完毕后可以链接到监控管理平台，但最主要的还是将日志传递到搜索引擎平台，进行索引、存储。最后用户从前端图形界面访问，

与搜索引擎关联，获取查询数据。在图形化展示方面，需要按照用户的条件组合对数据进行检索展示，固化常用的报表。

日志集中管理架构的组成模块是固定的，但架构本身是弹性的，随着 IDC 物理环境的变化而变化，在每一个环节上，分散的 agent、分布式的消息队列、过滤器可以分区域部署、自伸缩扩容，最终的数据流向是存储到唯一的搜索引擎以供用户查询。

### 12.1.3 日志集中框架

#### 1. Flume

Flume 是 Cloudera 为分布式环境下的日志收集而开发的，提供了一个高可用、高可靠的海量日志采集、聚合和传输的工具。Flume 目前有两个版本：Flume 0.9X 版本统称为 Flume-og，Flume 1.X 版本统称为 Flume-ng。Flume 初始的发行版本目前被统称为 Flume OG (Original Generation)，所有权归属于 Cloudera。因为 Flume 试图广泛地囊括和兼容输入、输出的各类接口，功能不断扩展，Flume OG 代码变得臃肿不堪、核心组件设计不合理、配置不标准等问题逐一暴露，在 Flume OG 的最后一个发行版本 0.94.0 中，日志传输不稳定的现象尤为严重。为了解决这些问题，2011 年 10 月 22 日，Cloudera 完成了 Flume-728，对 Flume 里程碑式地进行改动：重构核心组件、核心配置及代码架构，重构后的版本被统称为 Flume NG (Next Generation)，随后不久 Flume 纳入到了 Apache 开源基金组织旗下，由 Cloudera Flume 改名为 Apache Flume。

Flume-ng 组件承担了多种角色，既可以是 Agent，也可以是 Filter。如图 12-2 所示是 Flume-ng 的一个最简架构图，方框内是一个完整的 Flume-ng 组件。Source 代表 Flume-ng 接收的输入，从 console（控制台）、RPC（Thrift-RPC）、text（文件）、tail（UNIX tail）、syslog（syslog 日志系统，支持 TCP 和 UDP 等两种模式）、exec（命令执行）等数据源中都可收集；Channel 是存储通道，它可以将日志事件存储到本地内存、远程消息队列、数据库中；最后，Sink 对应着 Flume-ng 的输出方式，它可以将日志事件输出到下一个 Agent 中，形成日志处理链，也可以存储到分布式文件系统中，例如 Hadoop 的 HDFS。在架构图中没有体现的是 Flume-ng 的解释器功能，它相当于一个过滤器，对日志事件内容进行格式化或者直接丢弃。

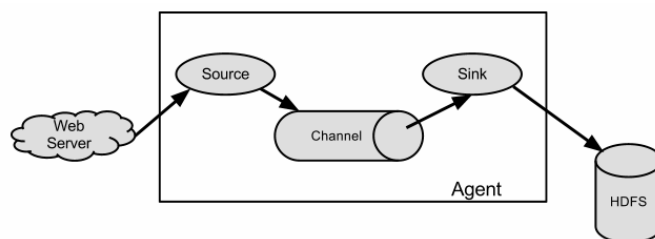


图 12-2 Flume-ng 简单架构图

正是由于 Flume 具备了输入、过滤、存储与输出功能，所以我们可以很快地用 Flume 构建一套分布式日志收集系统，如图 12-3 所示，Agent1~3 直接面向外部收集日志信息，而 Agent4 汇聚各方消息，过滤完成后将其存储到 HDFS 中。如图 12-3 所示。

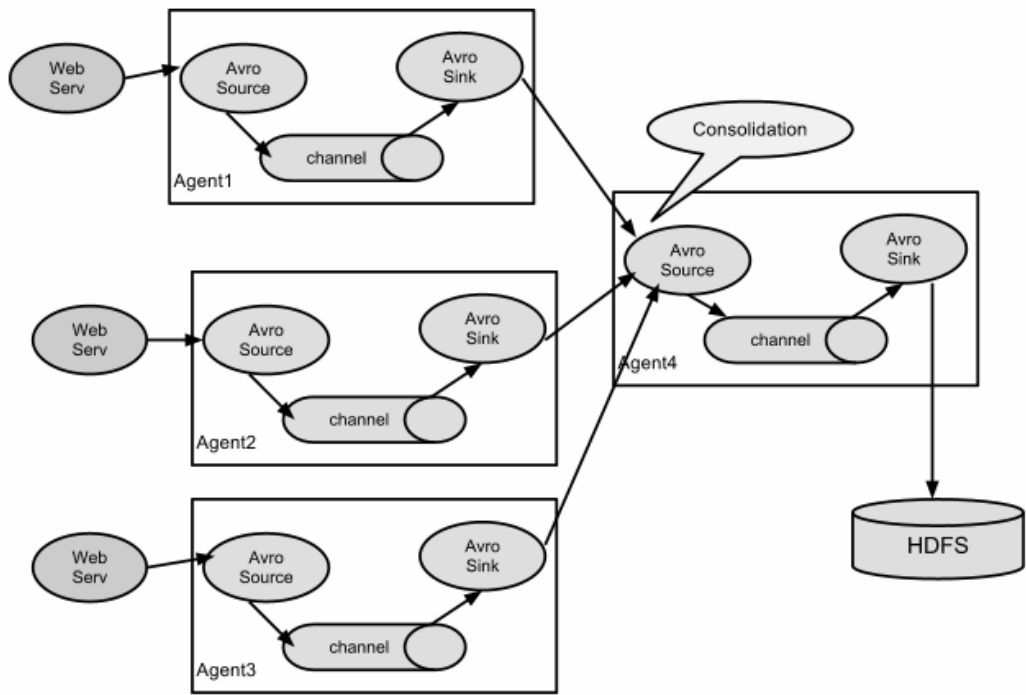


图 12-3 Flume 构建的分布式日志系统

Flume 在 Aapche 的 Hadoop 大数据生态圈中得到了广泛应用。著名的 Twitter 公司使用 Flume 将数据采集到 Hadoop HDFS 分布式存储中，之后使用 Hive 对数据进行分析。在 Flume 中每段数据都被称为事件，Source 负责生成事件，并连接 Source 与 Sink 的通道传递事件。Sink 负责把事件写入预定义的位置，最后将数据写入 HDFS 文件。Flume 在 Hadoop 生态圈的使用架构如图 12-4 所示。

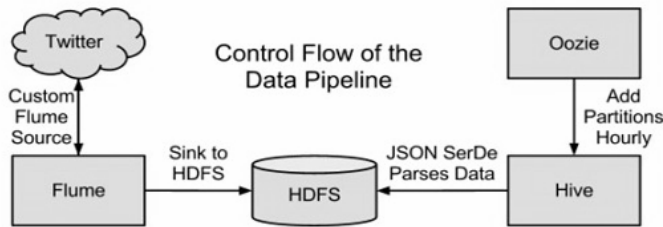


图 12-4 Flume 与 Hadoop 的生态圈

2. Splunk

Splunk 是一家总部位于美国加利福利亚州的跨国公司，其提供商用日志集中管理系

统，通过 Web 界面搜索、监控与分析机器产生的大数据。Splunk 的强大在于其对非结构化数据的分析功能。在系统运维领域，很多早期的工具聚焦在日志收集、集中上，而对于逐渐形成的大数据并没有一套成形的数据处理方法。Splunk 抓住了市场上的这个痛点，在数据检索、数据建模、图形展示上大下功夫，从而在业内处于领导地位，独树一帜。

Splunk 在架构上与其他日志集中管理平台基本一致，其最大的优势在于从起初的数据收集到最后的数据展示的整个处理过程都由一个产品提供服务，模块间的衔接在同一个商用解决方案内部。除此之外，在数据分析与展示上，Splunk 会提供行业内的业务数据模型，培训用户使用日志数据的方法。

Splunk 在数据查询上定义了一套内部的查询处理语言（Search Processing Language），搜索功能设定得非常简单，除了一般的关键字查找，还支持正则表达式抽取字段进行分组、聚合操作。支持类似于 UNIX 管道的方式将一个结果作为下一次查询的输入。它还提供大量的函数及自定义函数以便于用户对查询结果的处理。在可视化功能上，可以按照用户的需求定义报表，支持命令行、函数方式创建。在图表方面有包饼图、柱状图、行列图及计量图等。

### 3. ELK

与商用的 Splunk 相比，在开源社区也形成了一套日志集中管理的端到端解决方案，人们将其称为 ELK，即 Elasticsearch、Logstash 与 Kibana 三者的结合。Elasticsearch 作为搜索引擎平台，对数据进行索引与存储，并提供 API 查询接口；Logstash 是日志收集、过滤与输出的核心组件，其功能与 Flume 类似；在图形展示上使用 Kibana 从 Elasticsearch 中检索数据。ELK 的日志集中管理框架并不限定在这三个组件上，Logstash 的输入接口兼容多种类型的输入方式，例如 syslog、消息队列等。本书的重点将放在 ELK 框架上。

## 12.2 Logstash

### 12.2.1 Logstash介绍

Logstash 项目诞生于 2009 年，它的作者乔丹·西塞（Jordan Sissel）当时是虚拟主机托管商 DreamHost 的系统管理员，工作的主要内容是运维。他发现自己的很多时间都投入在与日志打交道中，于是构思了一种日志集中管理的方法。乔丹·西塞原本计划在 Logstash 内部实现一个简单的存储系统，当时 Elasticsearch 早已存在，但乔丹·西塞没有第一时间使用它。随着 Logstash 的流行，在开源社区的鼓动下，西塞将其与 Elasticsearch 结合在了一起，发现它是 Logstash 存储日志数据的最佳选择。2013 年乔丹·西塞加入 Elasticsearch。Logstash 作为 ELK 栈中的核心组件在不断成熟，并逐渐从运维日志管理转向业务数据分析



转型。Elasticsearch 本身也是近两年最受关注的大数据项目之一，三次融资已经超过一亿美元。在 Elasticsearch 开发人员的共同努力下，Logstash 的发布机制、插件架构也愈发科学合理。

“Logstash 可以从未知位置和任意源头获取数据，并将其格式化。你不用担心获取的日志类型，以及如何使其格式一致。”西塞说，“我们负责处理整个过程，使用 Elasticsearch 来索引与存储数据，通过 Kibana 让数据可视化。在关于如何让用户满意及驱动商业成功上，你可以立即得到解决方案。”

## 12.2.2 快速安装

### 1. 安装 JVM

Logstash 采用 JRuby 语言编写，它依赖于 Java 虚拟机，在安装 Logstash 之前我们必须先安装 Java 虚拟机。Java 虚拟机的安装方式有多种，可以直接使用 Linux 包管理器安装，也可以直接下载二进制可执行文件。

在 Ubuntu 上的安装方式如下：

```
root@ubuntu:~# apt-get install openjdk-7-dk
```

一旦完成了 JDK 的安装，我们就可以继续下面的步骤。

### 2. 环境变量

如前所述，Logstash 是一个 Java 进程，在启动一个 Java 进程之前我们可以设置相关的 Java Option 参数来调整虚拟机的设置，例如内存大小、线程栈空间等。Logstash 通过一个脚本启动，其中包括了默认 Java Option 参数的设置，对于需要新增参数的情况，我们可以直接在脚本外部设置环境变量 LS\_JAVA\_OPTS，脚本代码会读取变量中的值，并把它添加到 Java Option 参数后面。如果要覆盖原有默认 Java Option 参数，则需要编辑脚本，进行内容替换。

### 3. 安装运行

下载 Logstash 二进制文件，解压后用最简配置运行：

```
root@ubuntu:~# curl -O https://download.elasticsearch.org/logstash/logstash-1.5.0.tar.gz
root@ubuntu:~# tar zxvf logstash-1.5.0.tar.gz
cd logstash-1.5.0
bin/logstash -e 'input { stdin { } } output { stdout { } }'
```

最简配置从标准输入读取数据，之后输出到标准输出，我们在命令行上敲入“hello world”，输出立即返回，内容包括一个时间戳、主机名及消息正文：

```
hello world
2015-07-15T12:38:00.417Z ubuntu hello world
```

下面修改配置文件，将输出格式转变为 JSON 格式：

```
bin/logstash -e 'input { stdin { } } output { stdout { codec => json } }'
```

输出效果如下：

```
hello world
{"message": "hello world", "@version": "1", "@timestamp": "2015-07-15T12:40:19.609Z", "host": "ubuntu" }
```

### 4. 命令行参数

通过 help 选项可以查看 Logstash 的命令行参数：

```
root@ubuntu:~ bin/logstash --help
```

#### 1) -f, --config CONFIG\_PATH

从指定的文件或者文件夹中加载配置文件，如果给出的是一个文件夹，那么文件夹中的所有文件将按照字母顺序读取，之后解析成一个单独文件。在这里也可以用 globs 模糊匹配方式指定文件。

#### 2) -e CONFIG\_STRING

将配置文件以字符串的形式输入，在 hello world 的例子中就是用这种方式配置的。

#### 3) -w, --filterworkers COUNT

设置过滤程序的线程数，默认是 1。

#### 4) --watchdog-timeout SECONDS

在 Filter 模块执行的超时时间内，若 Filter 工作线程执行一个任务耗费很长时间，则一般可以认为是 Bug 导致，需要有一个超时退出机制，在这里默认时间是 10 秒。

#### 5) -l, --log FILE

设置 Logstash 自己的日志输出文件，如果没有设置，则该日志将被忽略。

#### 6) -v

显示详细的 Logstash 日志信息：指定一个 ‘v’ 代表 Info 级别日志；指定 ‘vv’，代表 Debug 级别的日志，与 --verbose 或 --debug 相同；--quiet 代表只输出 Error 级别的日志。

#### 7) -V, --version

显示当前 Logstash 的版本信息。

#### 8) -t, --configtest

对配置文件进行检查。

## 5. 插件

我们从 Logstash 命令行 -e 选项输入的配置信息中获知，内容分为了 input 与 output 两段，在后面的内容中还会包括 Filter。每一个段的处理方式是交由插件负责的，例如前面的 Stdin 与 Stdout 都是插件。

我们可以用 plugin 命令的 List 选项查询这两个插件：

```
root@ubuntu:~ bin/plugin list | grep -P 'stdin|stdout'
logstash-input-stdin
logstash-output-stdout
```

Logstash 的插件是通过 Gem 进行安装的。Gem 是一个管理 Ruby 库和程序的标准包，它通过 Ruby Gem（如 <http://Rubygems.org/>）源来查找、安装、升级和卸载软件包，非常便捷。

例如安装 Kafka Output 插件：

```
bin/plugin install logstash-output-kafka
```

指定文件路径安装：

```
bin/plugin install /path/to/logstash-output-kafka-1.0.0.gem
```

卸载插件：

```
bin/plugin uninstall logstash-output-kafka
```

升级插件：

```
bin/plugin update logstash-output-kafka
```

列出插件：

```
bin/plugin list
bin/plugin list [namefragment]
bin/plugin list --group output
```

### 12.2.3 配置说明

在配置文件中我们定义了 Logstash 设置方式，基本的配置分为：段（input、output、filter）、段中的插件设置（stdin、file、kafka 等）、插件所需要的参数。在段中还可以使用条件语句对消息内容进行判断、处理。

接下来看看下面的配置文件示例，它从本地标准输入读取数据，之后输出到 Elasticsearch 和标准输出中：

```
input { stdin { } }
output {
  elasticsearch { host => localhost }
```

```
    stdout { codec => rubydebug }
  }
```

## 1. 配置文件结构

Logstash 的配置文件是 JSON 格式的，内容分为多个段，在每一个段中可以加入一个或多个插件，在插件中可以配置相应的参数。在 `filter` 段中，如果有多个 `filter` 插件，则将按照它们在配置中出现的顺序进行处理。

```
# This is a comment. You should use comments to describe
# parts of your configuration.
input {
  ...
}

filter {
  ...
}

output {
  ...
}
```

## 2. 插件

插件内容包含在段中，每一个插件包括一个插件名及一组相关的设置参数。下面的示例为一个 `file` 插件从文件中读取数据，它有两个参数：`path` 指定日志文件路径，`type` 标明日志类型。

```
input {
  file {
    path => "/var/log/messages"
    type => "syslog"
  }

  file {
    path => "/var/log/apache/access.log"
    type => "apache"
  }
}
```

不同的插件有不同的设置参数，在后面的配置信息中会详细介绍。

## 3. 变量类型与注释

如同编程语言一样，插件的设置参数由不同的变量构成，这些参数属于一个具体类型，例如 `Array`、`Boolean`、`String`、`Number`、`Hash` 等。

## 1) Array

数组可以由单个字符串或多个字符串组成，如果同一个变量名被多次设置参数，则它将自动设置到一个数组中。

例如：

```
path => [ "/var/log/messages ", "/var/log/*.log " ]
path => " /data/mysql/mysql.log "
```

`path` 变量被设置成一个数组，第一行有两个值，第二行再次设置 `path` 变量，此时在数组内有三个字符串。

## 2) Boolean

一个 Boolean 变量（布尔变量）必须是 `true` 或者 `false`，注意 `true` 或者 `false` 关键字并不包含在引号内。

例如：

```
ssl_enable => true
```

## 3) String

一个字符串由一个字符序列组成，其包含在引号内。

例如：

```
name => "Hello world "
```

## 4) Number

Number 型变量由数组组成，其是浮点数或整数。

例如：

```
port => 33
```

## 5) Hash

Hash 型变量（哈希型变量）由一组 `key`、`value` 对组成，其格式是 “`field1`” => “`value1`”。使用空格对多个 `key`、`value` 键值对进行分割。

例如：

```
match => {
  "field1" => "value1"
  "field2" => "value2"
  ...
}
```

注释方法与 Perl、Ruby、Python 等脚本语言类似，通过 # 号进行注释。

例如：

```
# this is a comment

input { # comments can appear at the end of a line, too
  # ...
}
```

#### 4. 依赖于事件的配置

Logstash 进程的处理流程分为 3 个阶段：input->filter->output。input 产生事件，filter 修改事件内容，output 将其输出到其他地方。

一个事件实际上是一条日志，或者说是一条消息。事件有自己的属性，或者说在这条日志消息中有很多字段。例如 Apache 的 access 日志有状态码（200、404）、请求路径、处理时长、客户端 IP 地址等字段，在 Logstash 中将这此字段称为“fields”。

在 Logstash 中有一些配置项专门是依赖于从事件中获得的这些 fields 的，而 input 是产生事件的源头，其中没有所谓的 field，因此这些配置参数在 input 段中并不生效，只能在 filter 及 output 中设置。事件的配置主要有输出格式化、条件语句两类，这些配置是针对 field 设置的。

##### 1) field 引用

input 中产生的一个事件，或者说一条日志，其内容可以格式化成一个 JSON 文档。例如下面是一条 Apache access 日志，格式从原来的纯文本（plain）格式转换为了 JSON：

```
{
  "agent": "Mozilla/5.0 (compatible; MSIE 9.0)",
  "ip": "192.168.24.44",
  "request": "/index.html",
  "response": {
    "status": 200,
    "bytes": 52353
  },
  "ua": {
    "os": "Windows 7"
  }
}
```

从 Plain 转换为 JSON 的方法有很多，我们可以在配置 Apache 日志格式时就定义成 JSON 输出，也可以在 Logstash 的 filter 段中加入特殊的解析方法，将其转换成 JSON。在这里我们首先认为这个事件的格式就是遵循 JSON 标准的。

在这个 JSON 文档中我们可以找到很多 field，例如 agent、ip 等，field 分为顶级 field 和内嵌 field，agent、ip、request 都是顶级 field，而 status、bytes 及 os 都在 field 中，因此是内嵌 field。

事件传输到 filter、output 段后，在段中可以使用相关的 field 语法来定位字段内容，

[fieldname]代表具体字段，例如[ip]代表了“192.16.8.24.44”，而[ua][os]代表“Windows 7”。

定义好从事件中定位 field 并获取其 value 有什么用处呢？在后面的输出格式化，以及条件判断中将会依据 field 的 value 对内容进行处理。

## 2) sprintf format

对事件内容的格式化在 Logstash 中被称为 sprintf format，它将遵循一定规则的参数传入插件处理函数中，在插件内部进行格式化。下面的示例是 statsd output 插件的 increment 选项设置。在“increment =>”右边的 value 中，传入的是 type 类型为 Apache 的事件，百分号加花括号中的内容是这样一种表示；它告知插件其中的内容是动态、可变的，在该示例中通过 field 引用关联到了 access 日志中的 HTTP 响应状态码。statsd 插件是用于数据统计的，increment 选项用于在默认周期内对特定字段进行计数，在本示例中即对不同状态码在默认周期内发生的次数进行计数：

```
output {
  statsd {
    increment => "apache.%{[response][status]}"
  }
}
```

在上例中将 field 引用作为动态内容传入，我们还可以通过 sprintf format 格式化时间戳，例如 output 插件采用 file，将内容写入文件中，文件名默认是静态固定的，我们需要采用一种动态机制，将日期时间作为文件名传入，随着时间的变化，写入文件也跟着变化，下面的例子将日志事件写入一个以 type 为名称、以日期为后缀是日期的文件中：

```
output {
  file {
    path => "/var/log/%{type}_%{+YYYY.MM.dd.HH}"
  }
}
```

## 3) 条件语句

如编程语言一样，你希望依据某些条件来进行 filter、output，Logstash 中可以提供这样的功能。条件语句的语法如下：

```
if EXPRESSION {
  ...
} else if EXPRESSION {
  ...
} else {
  ...
}
```

语法中的 EXPRESSION 是条件判断的表达式，这些表达式有布尔逻辑、测试等，这些操作如下。

- 相等式：==、!=、<、>、<=、>=
- 正则表达式：=~、!~
- 包含式：in、not in
- 布尔操作：and、or、nand、xor
- 一元运算：!

表达式可能会很长，并且很复杂，表达式内部可以嵌入其他表达式，你可以通过！操作取反，也可以通过圆括号进行分组。

下面的例子判断 action 字段的内容是否是 login，如果是，则使用 mutate filter 插件移除 secret 字段的内容：

```
filter {
  if [action] == "login" {
    mutate { remove => "secret" }
  }
}
```

你能够在单行内指定多个表达式：

```
output {
  # Send production errors to pagerduty
  if [loglevel] == "ERROR" and [deployment] == "production" {
    pagerduty {
      ...
    }
  }
}
```

我们可以使用 in 操作符在条件中对值进行判断：

```
filter {
  if [foo] in [foobar] {
    mutate { add_tag => "field in field" }
  }
  if [foo] in "foo" {
    mutate { add_tag => "field in string" }
  }
  if "hello" in [greeting] {
    mutate { add_tag => "string in field" }
  }
  if [foo] in ["hello", "world", "foo"] {
    mutate { add_tag => "field in list" }
  }
  if [missing] in [alsomissing] {
    mutate { add_tag => "shouldnotexist" }
  }
}
```



```

    if !("foo" in ["hello", "world"]) {
      mutate { add_tag => "shouldexist" }
    }
  }
}

```

你可以使用 `not in` 操作符进行取反判断，在下例中我们使用 `not in` 来判断路由事件中被 `grok` 插件正确处理的事件，并将其输出到 Elasticsearch 中：

```

output {
  if "_grokparsefailure" not in [tags] {
    elasticsearch { ... }
  }
}

```

#### 4) @metadata 元数据字段

元数据是用来描述有效数据的数据，一个文本文件路径、大小、创建修改时间等都是描述这个文本文件本身的数据，而文本中的内容是有效数据，记录的是用户关心的信息。在 `logstash` 中元数据的 `key` 前用 “@” 符号标识，在 1.5 版本及之后的版本中加入了 `@metadata` 字段，专门用于让用户自己定义元数据。这些内容不会在最终的 `output` 插件中输出，使用 `@metadata` 的最大作用是在条件判断时加入一些中间变量标识。

下面的例子从 `stdin` 标准输入中采集日志信息，之后通过 `mutate filter` 插件进行处理，`mutate` 会加入三个 `field`，其中有两个是元数据 `field`：

```

input { stdin { } }

filter {
  mutate { add_field => { "show" => "This data will be in the
output" } }
  mutate { add_field => { "[@metadata][test]" => "Hello" } }
  mutate { add_field => { "[@metadata][no_show]" => "This data will
not be in the output" } }
}

output {
  if [@metadata][test] == "Hello" {
    stdout { codec => rubydebug }
  }
}

```

我们看到在 `filter` 中加入了 `@metadata` 的 `field` 有 `test`、`no_show`。在 `output` 插件中对 `test` 元数据进行判断，如果内容是 “Hello”，则将输出格式化为 `rubydebug` 类型。让我们看看输出内容：

```

$ bin/logstash -f ../test.conf
Logstash startup completed
asdf
{

```

```
"message" => "asdf",
"@version" => "1",
"@timestamp" => "2015-03-18T23:09:29.595Z",
"host" => "example.com",
"show" => "This data will be in the output"
}
```

启动 Logstash 后在标准输入中敲入“asdf”，在最后的输出中“asdf”是 message 字段的内容，条件语句中判断出了 @metadata 元数据的 test 字段，最后的输出格式是 rubydebug 类型。我们看到 @metadata 的数据信息并没有出现在最终输出中。

接下来修改配置文件，在 rubydebug 格式输出中运行 @metadata 数据输出：

```
stdout { codec => rubydebug { metadata => true } }
```

再看看输出内容：

```
$ bin/logstash -f ../test.conf
Logstash startup completed
asdf
{
  "message" => "asdf",
  "@version" => "1",
  "@timestamp" => "2015-03-18T23:10:19.859Z",
  "host" => "example.com",
  "show" => "This data will be in the output",
  "@metadata" => {
    "test" => "Hello",
    "no_show" => "This data will not be in the output"
  }
}
```

现在我们可以看见 @metadata 的内容了。

@metadata 中的字段内容一般用作中间过程的临时记录字段，在最终的输出中并不需要体现。如前所述，最终输出到 Elasticsearch 的事件格式是 JSON 的，它有一些默认的元数据字段，如 @timestamp，我们使用 @metadata 最可能的场景就是对于某些服务器的 access 日志，我们希望 @timestamp 中记录的日期是 HTTP 服务器 access 日志中的时间，而不是事件进入到 Logstash 的时间：

```
input { stdin { } }

filter {
  grok { match => [ "message", "%{HTTPDATE:[@metadata][timestamp]}" ] }
  date { match => [ "[@metadata][timestamp]", "dd/MMM/yyyy:HH:mm:ssZ" ] }
}

output {
```

```
    stdout { codec => rubydebug }
}
```

在上面的示例中，使用 `grok` 插件将 HTTP 日志的时间提取出来放入元数据 `@metadata[timestamp]` 中，再使用 `date` 插件读取这个元数据，对其进行日期格式化，之后覆盖 `@timestamp` 字段。下面看看 Logstash 的内容输出：

```
$ bin/logstash -f ../test.conf
Logstash startup completed
02/Mar/2014:15:36:43 +0100
{
  "message" => "02/Mar/2014:15:36:43 +0100",
  "@version" => "1",
  "@timestamp" => "2014-03-02T14:36:43.000Z",
  "host" => "example.com"
}
```

在介绍完 Logstash 的配置方法后，本节将以一个 Apache HTTP 服务器日志和 syslog 消息日志为例，完成一个 Logstash 配置的全过程。

## 5. 配置 filter 段

首先配置 Logstash 中的 filter 段，filter 能够对日志事件进行修改，在这个 filter 段中我们添加了 `grok`、`date` 两个处理插件：

```
input { stdin { } }

filter {
  grok {
    match => { "message" => "%{COMBINEDAPACHELOG}" }
  }
  date {
    match => [ "timestamp" , "dd/MMM/yyyy:HH:mm:ss Z" ]
  }
}

output {
  elasticsearch { host => localhost }
  stdout { codec => rubydebug }
}
```

用这个配置文件运行 Logstash：

```
bin/logstash -f logstash-filter.conf
```

在标准输入的控制台输入以下内容作为日志消息：

```
127.0.0.1 - - [11/Dec/2013:00:01:45 -0800] "GET /xampp/status.php HTTP/
1.1" 200 3891 "http://cadenza/xampp/navi.php" "Mozilla/5.0 (Macintosh;
Intel Mac OS X 10.9; rv:25.0) Gecko/20100101 Firefox/25.0"
```

我们将在标准输出中看到以下内容：

```
{
  "message" => "127.0.0.1 - - [11/Dec/2013:00:01:45 -0800] \"
GET /xampp/status.php HTTP/1.1\" 200 3891 \"
http://cadenza/xampp/navi.php\" \" Mozilla/5.0 (Macintosh; Intel Mac OS X
10.9; rv:25.0) Gecko/20100101 Firefox/25.0\" \" ,
  "@timestamp" => "2013-12-11T08:01:45.000Z" ,
  "@version" => "1" ,
  "host" => "cadenza" ,
  "clientip" => "127.0.0.1" ,
  "ident" => "-" ,
  "auth" => "-" ,
  "timestamp" => "11/Dec/2013:00:01:45 -0800" ,
  "verb" => "GET" ,
  "request" => "/xampp/status.php" ,
  "httpversion" => "1.1" ,
  "response" => "200" ,
  "bytes" => "3891" ,
  "referrer" => "\"http://cadenza/xampp/navi.php\" \" \" ,
  "agent" => "\"Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9;
rv:25.0) Gecko/20100101 Firefox/25.0\" \" \"
}
```

Logstash 输出的格式是一个 JSON 文档，我们可以看到通过对 `grok` 插件的处理，Apache 的一行日志被分割成了多个 `field` 字段，这样按照字段将数据保存到搜索引擎时，对后续的数据查询与分析有很大的帮助。例如用户希望通过 `HTTP` 状态码、来源 `IP` 地址等方式对数据进行检索。关于 `grok filter` 插件的使用方法将在后面进行介绍。

在这里使用的第二个 `filter` 插件是 `data`，多个插件在 `filter` 段中是按照顺序执行的，在经过 `grok` 处理之后，流入 `data` 之前，日志内容有一个被 `grok` 解析、分割出来的 `timestamp` 字段，它是 Apache HTTP 服务器中记录的日志时间。而 Logstash 中原始的 `@timestamp` 字段中记录的是这条日志进入 Logstash 时开始处理的时间，`data` 插件将 `@timestamp` 的值替换为与 `timestamp` 一致的值。

## 6. 处理 Apache 日志

接下来让我们修改 Logstash 配置文件，让其从文件系统中读取 Apache 日志信息，另外在 `filter` 中加入一些条件判断：

```
input {
  file {
    path => "/tmp/access_log"
    start_position => "beginning"
  }
}

filter {
```

```

if [path] =~ "access" {
  mutate { replace => { "type" => "apache_access" } }
  grok {
    match => { "message" => "%{COMBINEDAPACHELOG}" }
  }
}
date {
  match => [ "timestamp" , "dd/MMM/yyyy:HH:mm:ss Z" ]
}

output {
  elasticsearch {
    host => localhost
  }
  stdout { codec => rubydebug }
}

```

创建一个名为/tmp/access\_log 的文件，在其中输入以下信息：

```

71.141.244.242 - kurt [18/May/2011:01:48:10 -0700] "GET /admin HTTP/1.1"
301 566 "-" "Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.2.3)
Gecko/20100401 Firefox/3.6.3"
134.39.72.245 - - [18/May/2011:12:40:18 -0700] "GET /favicon.ico HTTP/1.1"
200 1189 "-" "Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/
4.0; .NET CLR 2.0.50727; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729; InfoPath.
2; .NET4.0C; .NET4.0E)"
98.83.179.51 - - [18/May/2011:19:35:08 -0700] "GET /css/main.css HTTP/1.1"
200 1837 "http://www.safesand.com/information.htm" "Mozilla/5.0 (Windows
NT 6.0; WOW64; rv:2.0.1) Gecko/20100101 Firefox/4.0.1"

```

文件中有三条日志信息，接着启动 Logstash：

```
bin/logstash -f logstash-apache.conf
```

我们看到 Apache 日志数据被发送到了 Elasticsearch 中。在配置文件的 filter 段条件判断逻辑中，它先通过正则表达式确认所处理文件的路径中包含了“access”，之后对这种类型的日志事件新增一个 type 字段，将其标识为“apache\_access”，最后引入一个 data 插件的时间戳修改。

第一次启动 Logstash 时会读取文件中的所有内容，之后它会保存一个当前文件所处理到的位置信息，在以后即便 Logstash 重启了，也是从新的日志内容开始处理。

我们再进行进一步：

```

input {
  file {
    path => "/tmp/*_log"
  }
}

```

```

filter {
  if [path] =~ "access" {
    mutate { replace => { type => "apache_access" } }
    grok {
      match => { "message" => "%{COMBINEDAPACHELOG}" }
    }
    date {
      match => [ "timestamp" , "dd/MMM/yyyy:HH:mm:ss Z" ]
    }
  } else if [path] =~ "error" {
    mutate { replace => { type => "apache_error" } }
  } else {
    mutate { replace => { type => "random_logs" } }
  }
}
output {
  if [type] == "apache" {
    if [status] =~ /^5\d\d/ {
      nagios { ... }
    } else if [status] =~ /^4\d\d/ {
      elasticsearch { ... }
    }
  }
  statsd { increment => "apache.%{status}" }
}
}

```

在上面的配置文件示例中，input 段的 file 插件使用了文件名匹配模式，以 log 结尾的所有日志文件都将被处理。在 filter 段中对路径进行正则表达式匹配，包含 access 的标识 type 为 “apache\_access”，error 的标识 type 为 “apache\_error”，其他的标识为 “random\_logs”。

output 段输出时将 HTTP 状态码为 5xx 的记录发送到 Nagios 告警系统中，将状态码是 4xx 的记录发送到 Elasticsearch 中，对其他的记录通过 statsd 插件进行数据统计。

## 7. 处理 Syslog 日志消息

Syslog 协议是在一个 IP 网络中转发系统日志信息的标准，它是在美国加州大学伯克利软件分布研究中心（BSD）的 TCP/IP 系统实施中开发的，目前已成为工业标准协议，可用它记录设备的日志。Syslog 记录系统中的任何事件，管理者可以通过查看系统记录随时掌握系统状况。系统日志通过 Syslog 进程记录系统的有关事件，也可以记录应用程序的运作事件。通过适当配置，还可以实现运行 Syslog 协议的机器之间的通信。通过分析这些网络行为日志，可追踪和掌握与设备和网络有关的情况。

接下来让我们建立一个示例配置文件来读取 Syslog 的输入：

```

input {
  tcp {
    port => 5000
  }
}

```

```

    type => syslog
  }
  udp {
    port => 5000
    type => syslog
  }
}

filter {
  if [type] == "syslog" {
    grok {
      match => { "message" => "%{SYSLOGTIMESTAMP:syslog_timestamp}
%{SYSLOGHOST:syslog_hostname} %{DATA:syslog_program}(?:\[ %{POSINT:syslog_
pid}\])?: %{GREEDYDATA:syslog_message}" }
      add_field => [ "received_at", "%{@timestamp}" ]
      add_field => [ "received_from", "%{host}" ]
    }
    syslog_pri { }
    date {
      match => [ "syslog_timestamp", "MMM d HH:mm:ss", "MMM dd HH:
mm:ss" ]
    }
  }
}

output {
  elasticsearch { host => localhost }
  stdout { codec => rubydebug }
}

```

用新配置文件来启动 Logstash:

```
bin/logstash -f logstash-syslog.conf
```

在配置文件的 `input` 段中有 `tcp`、`udp` 两个插件，其实它们仅仅简单监听 5000 端口，并将内容的 `type` 标识为 Syslog。

我们尝试着 telnet 这个端口：

```
telnet localhost 5000
```

复制下面的内容到 telnet 命令标准输入中：

```

Dec 23 12:11:43 louis postfix/smtpd[31499]: connect from unknown[95.
75.93.154]
Dec 23 14:42:56 louis named[16000]: client 199.48.164.7#64817: query
(cache) 'amsterdamboothuren.com/MX/IN' denied
Dec 23 14:30:01 louis CRON[619]: (www-data) CMD (php /usr/share/cacti/
site/poller.php >/dev/null 2>/var/log/cacti/poller-error.log)
Dec 22 18:28:06 louis rsyslogd: [origin software="rsyslogd" swVersion="
4.2.0" x-pid=" 2253" x-info="http://www.rsyslog.com"] rsyslogd was

```

HUPed, type 'lightweight'.

经过 filter 中各个插件的处理, 输出结果如下:

```
{
    "message" => "Dec 23 14:30:01 louis CRON[619]: (www-
data) CMD (php /usr/share/cacti/site/poller.php >/dev/null 2>/var/log/
cacti/poller-error.log) ",
    "@timestamp" => "2013-12-23T22:30:01.000Z",
    "@version" => "1",
    "type" => "syslog",
    "host" => "0:0:0:0:0:0:0:1:52617",
    "syslog_timestamp" => "Dec 23 14:30:01",
    "syslog_hostname" => "louis",
    "syslog_program" => "CRON",
    "syslog_pid" => "619",
    "syslog_message" => "(www-data) CMD (php /usr/share/cacti/
site/poller.php >/dev/null 2>/var/log/cacti/poller-error.log)",
    "received_at" => "2013-12-23 22:49:22 UTC",
    "received_from" => "0:0:0:0:0:0:0:1:52617",
    "syslog_severity_code" => 5,
    "syslog_facility_code" => 1,
    "syslog_facility" => "user-level",
    "syslog_severity" => "notice"
}
```

## 12.2.4 部署架构

由于 Logstash 本身具备 input、output 及 filter 功能, 它在日志集中管理中可以充当多种角色, 即有多种部署架构, 下面从最简方式开始介绍。

Logstash 作为唯一的模块, 从数据源收集、过滤数据, 最后输出到 Elasticsearch 中。如图 12-5 所示。

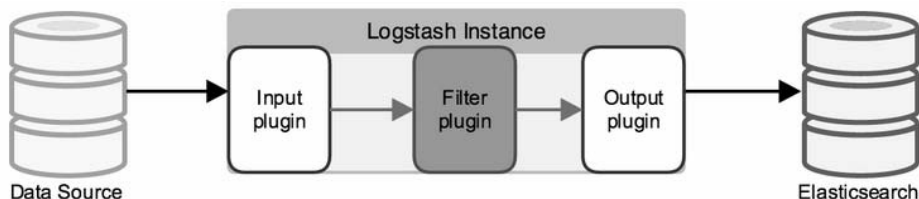


图 12-5 Logstash 作为唯一模块的日志集中架构

在一个 Logstash Instance 内部, 事件从一个阶段传递到下一个阶段, 从 Input 到 Filter, 再到 Output, 在每个阶段之间将使用 Ruby 语言实现的内部队列作为缓存, 每个队列的长度是 20, 这意味着进入下一个阶段可缓存的最大队列长度是 20, 中间缓存的作用是防止下一个阶段的处理过慢导致上一个阶段处理完的事件丢失或者等待。



当进入下一个阶段的队列满了的情况下，上一个阶段的任务将被堵塞，等待队列空闲。例如某一个 `filter` 段的插件在处理消息时过慢，那么将影响到前端 `input` 消息的接收，在 20 个消息排满后，前端不再接收外部请求。在 `Logstash` 的启动参数中可以设置 `filter` 插件处理的超时时间，以防队列堵塞。并不是说将内部缓存消息队列长度调整得越大控制队列数确保的实例就会运行得越稳定，但在繁忙阶段，处理效能有较大出入时，会发生事件消息丢失的情况。假如不对队列长度加以限制，则在这种繁忙阶段出现的情况可能就不是消息丢失了，而是整个实例因为内存超限而直接崩溃。

`Logstash` 三个段中的线程处理模型是不一样的。对于 `input` 段，其中的每一个插件会启动一个独立的 `thread`，这样可以避免处理繁忙的插件影响其他插件。对于 `filter` 段，其中包含了次序排列的多个插件及条件处理逻辑；`filter` 采用多线程工作模式，每一个线程的处理逻辑全部一致，包含了所有的插件，默认 `filter` 的工作线程是 1，通过 `Logstash` 启动参数可加大线程数，提升处理性能。最后的 `output` 段是单线程处理模式，因此我们可以看到，如果将三个段的功能全部放在一个实例中，则 `output` 作为处理链上的最后一个段很可能成为性能处理短板。当然，有的 `output` 插件在这方面做了考量与优化，它并不是一条一条地处理消息，对于外部输出是一个请求提交多条消息。例如 `Elasticsearch` 的 `output` 插件，它并不是将一条条消息写入 `Elasticsearch` 节点，而是将多条消息作为一个请求进行提交。

为了提高吞吐量，我们部署多个 `Logstash` 实例，此时由于有多个实例收集并向 `Elasticsearch` 索引数据，所以性能大幅提升。如图 12-6 所示。

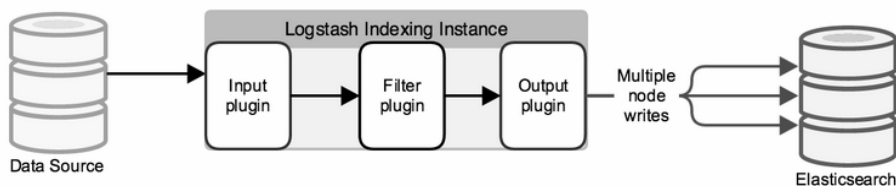


图 12-6 Logstash 多节点写入 ES

当 `input` 的数据源的日志消息越来越大，接收消息的数据速度超过了 `Logstash` 后端 `filter` 处理、索引建立 `output` 处理时，日志消息会出现丢失状态，此时可以将 `Logstash` 实例拆分成 `shipping` 与 `indexing` 两个角色，并在二者之间放入消息队列作为缓存，`shipping` 负责接收日志数据并将其快速地放入消息队列中，`indexing` 则从消息队列中获取数据并进行索引处理，最后存放到 `Elasticsearch` 中。如图 12-7 所示。

最后，我们将 `Logstash` 的 `shipping` 功能扁平化，每一个 `shipping` 具备相同的 `input` 处理功能，所有类型的日志数据可以任意地发送到一个 `shipping` 中进行处理。如图 12-8 所示的文件、UDP 网络及 RSS 订阅协议的数据全部被发送到 `Logstash shipping` 中。

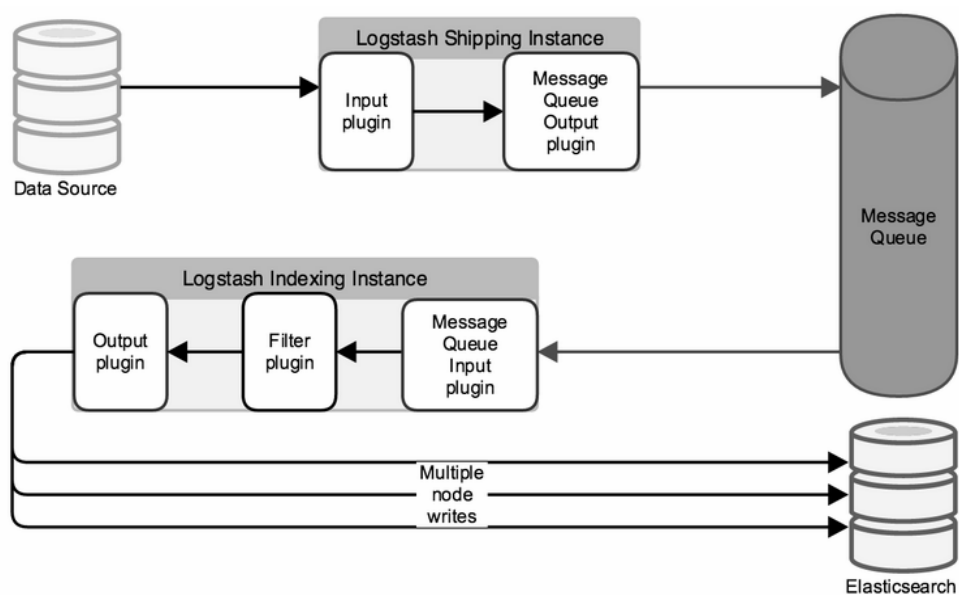


图 12-7 增加消息队列的 Logstash 多节点写入

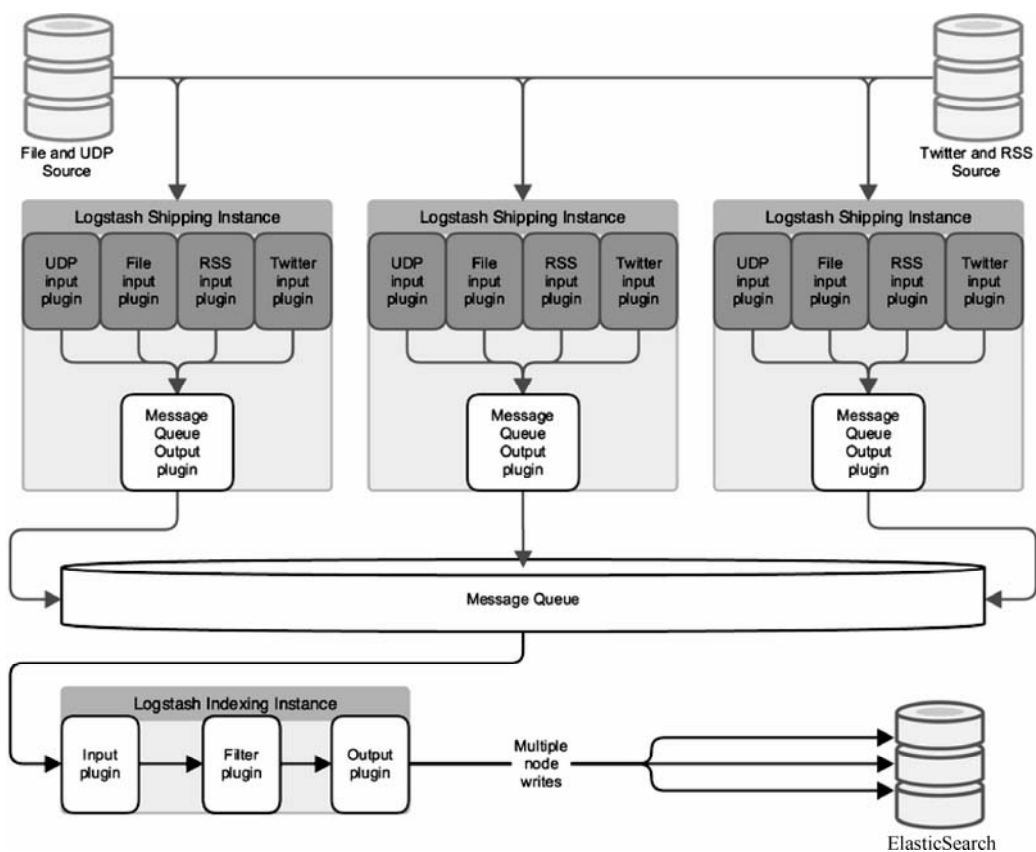


图 12-8 完整的 Logstash 日志架构

在整个部署架构中可以随意调整 Logstash，增加节点，提升性能。架构中的消息队列及 Elasticsearch 同样具备强伸缩性。

## 12.2.5 处理流程

Logstash 的日志事件处理方式遵循了三阶段方式：input->filter->outputs。input 生成数据，filter 修改内容，output 将格式化后的内容发送到其他地方。

### 1. input

我们使用 input 段来收集数据，Logstash 中一些常用的插件如下。

- **File**: 从文件系统中读取文件，类似于 UNIX 系统命令 `tail -0a`。
- **Syslog**: 监听 514 端口，等待从远端发送过来的 Syslog 日志。
- **Redis**: Redis 是开源社区流行的内存数据库，其常被用来作为 Logstash 的消息队列使用，Logstash 使用 redis 的 channels、lists 方式从 Redis 中读取数据。
- **Lumberjack**: Lumberjack 是轻量级的日志收集器，也被称为 Logstash-forwarder，为了提高本地文件的日志收集性能，常常使用 Lumberjack 替代 Logstash 的 input，之后将消息输出到 Logstash 中。

### 2. filter

filter 是 Logstash 处理流程的中间环节，在 filter 段，我们可以使用条件判断语句来实现某些动作，下面是常用的 filter。

- **grok**: 对文本内容进行解析与结构化，grok 是当前 Logstash 中用来将非结构化数据转为可查询、结构化的最好方式，有 120 多种默认的匹配方式内嵌在 Logstash 中，基本可以满足我们所有的格式需求。
- **mutate**: 实现一些一般的事件字段转换方法，我们能够重命名、删除、替换，以及修改事件中的字段内容。
- **drop**: 直接将事件丢弃，例如 debug 信息的直接丢弃。
- **clone**: 将事件进行克隆，并添加、删除一些字段。
- **geoip**: 添加依据 IP 地址关联到的地理位置信息，在后续的 Kibana 图形展示中将用到。

### 3. output

output 是 Logstash 工作流的最后阶段，一个事件能够发送到多个 output，只有所有的 output 处理过程完成后，事件的执行才真正完毕。下面是一些常用的 output。

- **elasticsearch**: 将事件数据发送到 Elasticsearch 中。
- **file**: 将事件数据写入到文件中。
- **statsd**: 是专用的数据统计插件，可以按照时间周期进行分组计数、求和等各类统计工作。

Logstash 在解析处理流程中引入了一个重要的功能——**codecs**。**codecs** 即编码（**encode**）、解码（**decode**）的意思。要理解 **codecs**，我们先从字符集编码说起，在计算机中所有的数据都是二进制的，如果要将二进制映射到字符上，则必须遵循字符编码，通用的字符集编码是 UTF-8，在收到二进制数据后用 UTF-8 来解码并找到对应的每一个字符。

**codecs** 与字符集编码在概念上是一致的，但其超越了字符集的范畴，是从文本语言格式上来进行编码的，例如 JSON、multiline、plain 等。

- **plain**: 类似于字符集编码，在 **input** 中接收的数据默认是用 UTF-8 字符集进行解析的，如果源端使用了其他字符集，那么在 **codecs** 的 **plain** 中可进行相关设置来完成解析。
- **JSON**: 将数据编码、解码成 JSON 格式；即便 **input** 的内容是以 JSON 格式书写的，logstash 也将其作为 **plain** 进行解析，因此所有的 JSON 内容都会放入 **message** 字段中，如果希望 Logstash 在 **filter** 阶段以 JSON 方式解析数据，那么在 **input** 中要使用 JSON 的 **codecs**。
- **multiline**: 一般意义上的日志信息是一行数据，作为一条消息发送到 Logstash，但对于 Java 异常堆栈信息，这类错误日志是以多行方式体现的，因此在介绍日志消息时要重新定义消息的分隔符，通过 **multiline codecs** 完成这个任务。

## 12.2.6 input 插件

最常用的 **input** 插件包括：作为采集器从本地文件中输入；兼容性地从 Syslog 中获取；在可扩展的架构中从消息队列中读取。本节我们主要介绍文件（**file**）、消息队列。

### 1. file

Logstash 使用一个名叫 FileWatch 的 Ruby Gem 库来监听文件变化。这个库支持 glob 展开文件路径，而且会记录一个叫作 **sincedb** 的数据库文件来跟踪被监听的日志文件的当前读取位置。所以不用担心 Logstash 会对文件数据做重复处理或者发生数据遗漏。

下面对 **file** 插件的主要参数进行介绍，其中 **path** 参数是必须填写的：

```
file {  
  path => ...  
}
```

file 插件的主要参数如表 12-1 所示。

表 12-1 file 插件的主要参数

参 数	类 型	是否必须	含 义	默 认 值
add_field	hash	No	在事件中添加新字段	{}
codec	codec	No	codec 设置	" plain "
delimiter	string	No	设置消息分隔符	" \n "
discover_interval	number	No	path 中使用 glob 模糊匹配时，定期发现新文件的频率	15
exclude	array	No	排除那些不想收集的日志文件	
path	array	Yes	收集日志文件的路径，支持 glob 模糊匹配	
sourcedb_path	string	No	设置记录日志文件处理当前位置的数据存放的地址	
sourcedb_write_interval	number	No	sourcedb 的写入频率	15
start_position	string, one of [ " beginning ", " end " ]	No	每次收集日志文件从什么地方开始的信息，有开头与末尾两种设置	" end "
stat_interval	number	No	检查日志文件状态变化的频率	1
tags	array	No	添加标签	
type	string	No	添加类型	

2. 消息队列

在部署架构中需要添加消息队列，将收集、处理与索引的 Logstash 实例进行解耦，提升整体性能。

1) redis

Logstash 最常使用的消息队列是 Redis，它支持从 list（一般的 key-value 缓存，通过 BLPOP 命令读取）、channel（消息监听，使用 SUBSCRIBE 或 PSUBSCRIBE 命令读取）两种类型的数据中读取。为了提升性能，batch\_count 参数使用 Redis 内置的 EVAL 命令获取多条事件。如果需要在 Logstash 中配置多个 Redis 源，则可以添加多个 Redis 插件段。下面是一个配置示例，其中有两个 Redis 源：

```
input {
  redis {
    host => "10.0.0.10"
    data_type => "list"
    type => "redis-input"
    key => "logstash"
  }
  redis {
    host => "10.0.0.11"
```

```
data_type => "list"
type => "redis-input"
key => "logstash"
}
}
```

具体的参数说明如表 12-2 所示。

表 12-2 Logstash 配置消息队列 Redis 参数

参 数	类 型	是否必须	含 义	默 认 值
add_field	hash	No	在事件中添加新字段	{}
batch_count	number	No	当使用 Redis 的 EVAL 命令时从队列中返回的事件数	1
codec	codec	No	codec 设置	" json "
data_type	string, one of [ " list ", " channel ", " pattern_channel " ]	No	获取事件的数据类型，list 对应 BLPOP，channel 对应 SUBSCRIBE，pattern_channel 对应 PSUBSCRIBE	
db	number	No	Redis 数据库的 number	0
host	string	No	Redis 数据库地址，它是一个字符型，如果需要从多个 Redis 中获取数据，则在 input 中放置多个 Redis 插件	" 127.0.0.1 "
key	string	No	Redis 中 list 或 channel 的 key 名称	
password	password	No	连接 Redis 的密码	
port	number	No	Redis 服务器的端口	6379
tags	array	No	添加标签	
threads	number	No	连接线程数	1
timeout	number	No	初始化连接超时时间	5
type	string	No	添加类型	

2) Kafka

Kafka 是 LinkedIn 用于日志处理的分布式消息队列，LinkedIn 的日志数据容量大，但对可靠性要求不高，其日志数据主要包括用户行为（登录、浏览、单击、分享、喜欢），以及系统运行日志（CPU、内存、磁盘、网络、系统及进程状态）。当前很多消息队列服务提供可靠交付保证，并默认是即时消费（不适合离线）。高可靠交付对 LinkedIn 的日志不是必需的，故可通过降低可靠性来提高性能，同时可构建分布式的集群，允许消息在系统中累积，使得 Kafka 同时支持离线、在线日志处理。

对于某些场景，我们可以牺牲可靠性来保证日志处理的性能，选择 Kafka 消息队列替换 Redis。

因为 Kafka 消息队列专门针对分布式场景使用，下面我们对其基本概念与组成架构做一个简要介绍。

Kafka 包括以下基本概念。

- **broker**: Kafka 集群包含一个或多个服务器, 这种服务器被称为 **broker**。
- **topic**: 每条发布到 Kafka 集群的消息都有一个类别, 这个类别被称为 **topic** (物理上不同 **topic** 的消息分开存储, 逻辑上一个 **topic** 的消息虽然保存于一个或多个 **broker** 上, 但用户只需指定消息的 **topic**, 即可生产或消费数据而不必关心数据存于何处)。
- **partition**: **partition** 是物理上的概念, 每个 **topic** 包含一个或多个 **partition**, 创建 **topic** 时可指定 **partition** 数量。每个 **partition** 对应于一个文件夹, 该文件夹下存储该 **partition** 的数据和索引文件。
- **producer**: 生产者, 负责发布消息到 Kafka **broker**。
- **consumer**: 消费者负责消费消息。每个 **consumer** 属于一个特定的 **consumer group** (可为每个 **consumer** 指定 **group name**, 若不指定 **group name**, 则为默认的 **group**)。使用 **consumer high level API** 时, 同一 **topic** 的一条消息只能被同一个 **consumer group** 内的一个 **consumer** 消费, 但多个 **consumer group** 可同时消费这一消息。

Kafka 集群在架构上使用 ZooKeeper 共享与同步配置信息, 消息队列的使用方 (生产者、消费者) 从 ZooKeeper 中读取集群中 Kafka 节点的信息, 之后与 Kafka **broker** 进行交互。如图 12-9 所示是 Kafka 集群的简易架构图, 集群中有 4 个 Kafka **broker**, 它们分布在不同的操作系统节点上, 在示例中有 **node1**、**node2**。在 Kafka 安装文件中自带了 ZooKeeper, 在启动 Kafka **broker** 之前, 首先要保证 ZooKeeper 启动, 之后 **broker** 启动, 它们将配置、状态信息同步到 ZooKeeper 中。随着生产者、消费者对消息队列进行使用, 它们会从 ZooKeeper 发现集群中有多少 Kafka **broker**, 以及它们的读写方式。对生产者而言, 这不仅可以从 ZooKeeper 中获取数据, 也可以直接连接到 **broker** 上发送消息。

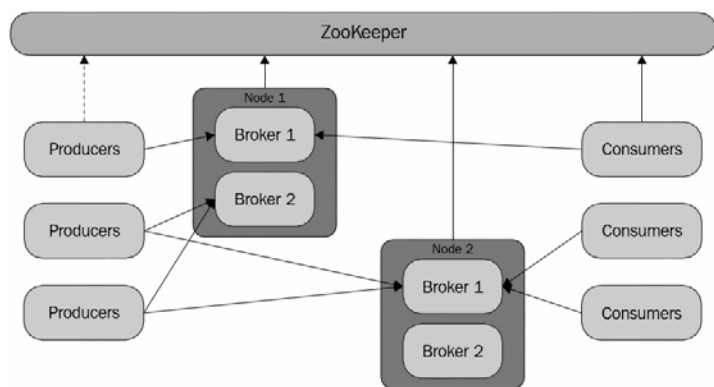


图 12-9 Kafka 集群简易架构图

对于日志集中管理的 Logstash 而言, **input** 是消费者, 从 **broker** 中获取消息, 而 **output**

扮演着生产者角色，将日志消息存储到 broker 中。

Kafka 保证在同一 consumer group 中只有一个 consumer 会消费某条消息，实际上，Kafka 保证的是稳定状态下将 partition 分配给唯一的 consumer，而每个 consumer 实例只会消费某个或多个特定 partition 的数据。这样设计将问题简单化，每个 consumer 不用跟大量的 broker 通信，减少通信开销，但无法让同一个 consumer group 里的 consumer 均匀消费数据。如果某 consumer group 中 consumer 的数量少于 partition 数量，则至少有一个 consumer 会会消费多个 partition 的数据，如果 consumer 的数量与 partition 数量相同，则正好一个 consumer 消费一个 partition 的数据，而如果 consumer 的数量多于 partition 的数量，则会有部分 consumer 无法消费该 topic 下的任何一条消息。

当 consumer group 中的节点发生变化时，例如增加、减少了 consumer，则会引起 Kafka 对 partition 的重新分配，这个动作被称为 rebalance。

Logstash 的 input kafka 插件配置示例如下：

```
input {
  kafka {
    zk_connect => "10.31.64.11:2181"
    group_id => "logstash"
    topic_id => "request"
    reset_beginning => false
    consumer_threads => 1
    codec => json {}
  }
}
```

具体配置信息如表 12-3 所示。

表 12-3 Logstash 的 input kafka 插件配置

参 数	类 型	是否必须	含 义	默 认 值
add_field	hash	No	在事件中添加新字段	{}
auto_offset_reset	string, one of [ "largest" , "smallest" ]	No	消息队列如同文件一样，内容有头、尾，这个配置用于判断没有设置队列偏移位置时，LogStash 是从消息的开头重新读取整个队列，还是从末尾读取最新消息	" largest "
black_list	string	No	对于集群中的某些 topic，如果不想去读取，则加入到此黑名单中	nil
codec	codec	No	codec 设置	" json "
consumer_id	string	No	唯一的消费者 ID，如果不设置，则自动生成	nil
consumer_restart_on_error	boolean	No	在消费者出现异常时是否重启	TRUE
consumer_restart_sleep_ms	number	No	异常重启的等待时间，单位为毫秒	0



续表

参 数	类 型	是否必须	含 义	默 认 值
consumer_threads	number	No	Kafka 的消息可以分为多个区，针对分区消费者可以设置多个读取线程。需要特别注意的是，当线程数多过分区数量时，这些多余的线程将一直保持空闲状态	1
consumer_timeout_ms	number	No	在一定的周期内没有消息可读时抛出一个超时异常	-1
decoder_class	string	No	定义消息序列化的处理类	"kafka.serializer.DefaultDecoder"
decorate_events	boolean	No	将 Kafka 的 topic、消息大小等信息作为元数据加入事件中	FALSE
fetch_message_max_bytes	number	No	设置获取消息的最大字节长度。这个长度应该不小于 Kafka 上设置的消息最大长度，否则会出现生产者发送的消息比消费者可以获取的长度要大。	1048576
group_id	string	No	在 Kafka 中将消费者进行了分组，该选项用来设置组名称	"logstash"
key_decoder_class	string	No	指定 key 序列化的类	"kafka.serializer.DefaultDecoder"
queue_size	number	No	Logstash 内部的队列长度，用来缓存从 Kafka 中读取的消息	20
rebalance_backoff_ms	number	No	当 consumer group 发生变化时，引发 rebalance、Logstash 定义的一次 rebalance 的完成时间	2000
rebalance_max_retries	number	No	rebalance 的重试次数	4
reset_beginning	boolean	No		FALSE
tags	array	No	添加标签	
topic_id	string	No	topic id	nil
type	string	No	添加类型	
white_list	string	No	对于集群中的某些 topic，将其加入到白名单后，Logstash 从该 topic 中获取消息	nil
zk_connect	string	No	集群中 ZooKeeper 的地址	"10.31.64.11:2181"

### 12.2.7 output插件

对于输出插件来说，如果是专门用于收集的 shipping Logstash 实例，则它们将信息发送到消息队列中。而对于最后的 indexing Logstash 实例，它们则将消息发送到 Elasticsearch，本节将重点介绍这两类 output 插件。

#### 1. Elasticsearch

Logstash 是 ELK stack 的关键一员，在日志最后的存储与索引上常常与同一公司旗下的全文搜索引擎 Elasticsearch 结合。Logstash 的 Elasticsearch 插件可以选择三种协议与后端通信，分别是 Node、HTTP 和 Transport。

在一个小集群里，使用 Node 协议是最为方便的。Logstash 以 Elasticsearch 的 client 节点身份（即不存数据不参加选举）运行，实际上是将 Logstash 作为 Elasticsearch 集群中的一员。对于某些在插件中无法配置的 Elasticsearch 节点参数，可以在 Logstash 的当前目录下创建 elasticsearch.yml 进行配置。Logstash 默认采用 Node 协议，我们需要保证它与 Elasticsearch 之间的 9300 端口双向可互相访问。

Logstash 相当于内嵌了一个 Elasticsearch 服务器，只是不存取数据而已，对于非生产环境，我们可以打开开关，将 embedded 参数设置为 true，让一个单一的 Logstash 既作为 indexing，又作为 Elasticsearch 服务存储数据。

示例配置如下：

```
output {
  elasticsearch {
    host => "10.11.77.40"
    workers => 5
  }
}
```

具体参数信息如表 12-4 所示。

表 12-4 Logstash 的 Elasticsearch 插件参数

参 数	类 型	是否必须	含 义	默 认 值
action	string	No	代表该 Logstash 操作 Elasticsearch 的方式，有 index 和 delete 两种选择	" index "
bind_host	string	No	作为 Elasticsearch 集群中一员时，选择的服务绑定地址	
bind_port	number	No	作为 Elasticsearch 集群中一员时，选择的服务绑定端口	
cacert	a valid filesystem path	No	用来做校验的以.cer 或.pem 为后缀的证书文件	

续表

参 数	类 型	是否必须	含 义	默 认 值
cluster	string	No	将要输出的 Elasticsearch 集群的名称。 当使用 Node、Transport 协议时，需要使用此名称来进行服务发现，默认为 Elasticsearch	elasticsearch
codec	codec	No	codec 设置	" plain "
embedded	boolean	No	是否将 Logstash 作为 Elasticsearch 服务器存储数据使用	FALSE
embedded_http_port	string	No	内嵌 Elasticsearch 服务的端口	" 9200-9300 "
flush_size	number	No	为了提升性能，在处理日志消息时以批量所有方式执行。该选项用来设置可缓存的事件数。	5000
host	array	No	输出的 Elasticsearch 服务器地址。如果是 Node 协议，未设置此选项，则将采用组播方式发现服务器	
idle_flush_time	number	No	当 flush_size 值在 idle_flush_time 内一直无法达到索引处理量时，强制刷新	1
index	string	No	写入 Elasticsearch 的 index 名称	" logstash-%{+YYYY.MM.dd} "
max_retries	number	No	当 index 出错时，重试的次数	3
password	password	No	连接 Elasticsearch 的密码	
port	string	No	Elasticsearch 所使用的端口，在默认情况下：protocol => http - port 9200； protocol => transport - port 9300-9305； protocol => node - port 9300-9305	
protocol	string, one of [ "node " , "transport " , "http " ]	No	选择协议类型	
retry_max_interval	number	No	当 index 出错时，重试的频率	5
retry_max_items	number	No		5000
ssl	boolean	No		FALSE
ssl_certificate_verification	boolean	No		TRUE
template_name	string	No		" logstash "

续表

参 数	类 型	是否必须	含 义	默 认 值
template_overwrite	Boolean	No		FALSE
user	string	No	用户名、密码设置，只对 HTTP 生效	
workers	number	No	设置 output 线程数量。前面讲过 output 是单线程模式，这里的 worker 只用来与 Elasticsearch 通信使用，并不是从 filter 队列中获取消息的	1

## 2. 消息队列

Redis 的 output 插件示例配置如下：

```
output {
  redis {
    host => [ "10.0.0.10", "10.0.0.11" ]
    shuffle_hosts => true
    data_type => "list"
    key => "logstash"
  }
}
```

在配置文件中 host 参数代表输出的远端 Redis 服务器；data\_type 代表存储的数据结构为 list，设置 shuffle\_hosts 为 true，代表着当 Logstash 重启后，只需要 host 列表中的任意 Redis 节点即可发送事件。

Redis 消息队列的 output 插件的具体配置参数如表 12-5 所示。

表 12-5 Redis 消息队列 output 插件的具体配置参数

参 数	类 型	是否必须	含 义	默 认 值
batch	boolean	No	是否允许批处理	FALSE
batch_events	number	No	批处理的事件量	50
batch_timeout	number	No	批处理的超时时间	5
codec	codec	No	codec 设置	" plain "
congestion_interval	number	No	拥塞检测的频率，默认为 1，代表每一个事件都做检查。	1
congestion_threshold	number	No	拥塞控制用来防止在事件量过多而导致 Redis 服务 OOM，但 reids list 超过此阈值时，Logstash 不再往里写数据，默认为开启，不做任何限制	0
data_type	string, one of [ " list ", " channel " ]	No	写入数据结构的类型，该选项必填	

续表

参 数	类 型	是否必须	含 义	默 认 值
db	number	No	Redis 数据库的 number	0
host	array	No	Redis 数据库服务的地址	[ " 127.0.0.1 " ]
key	string	No	写入 Redis 中的数据 key 值，该选项必填	
password	password	No	Redis 服务器的密码	
port	number	No	Redis 服务器的端口	6379
reconnect_interval	number	No	重连频率	1
shuffle_hosts	boolean	No	当 Logstash 重启后只需要 host 列表中的任意一 Redis 节点即可发送事件	TRUE
timeout	number	No	初始化超时时间	5
workers	number	No	工作进程数	1

Kafka output 消息队列 output 插件的具体参数如表 12-6 所示。

表 12-6 Kafka output 插件具体参数

参 数	类 型	是否必须	含 义	默 认 值
batch_num_messages	number	No	批处理方式的事件数	200
broker_list	string	No	broker 清单	" 10.31.64.11:9092 "
client_id	string	No	定义客户端名称，用来后续做问题跟踪	" "
codec	codec	No	codec 设置	" json "
compressed_topics	string	No	是否进行消息压缩	" "
compression_codec	string, one of[ " none ", " gzip ", " snappy " ]	No	压缩的格式	" none "
key_serializer_class	string	No	key 序列化类	" kafka.serializer.StringEncoder "
message_send_max_retries	number	No	当消息发送出错后的重试次数	3
partition_key_format	string	No	进行消息分区的 key 值	nil
partitioner_class	string	No	进行消息分区的处理类	" kafka.producer.DefaultPartitioner "
producer_type	string, one of[ " sync ", " async " ]	No	消息传送的方式，同步或异步	" sync "

续表

参 数	类 型	是否必须	含 义	默 认 值
queue_buffering_max_messages	number	No	未发送的最大消息队列数量	10000
queue_buffering_max_ms	number	No	最大缓存时间，单位为毫秒	5000
request_required_acks	string, one of [-1, 0, 1]	No	但发送消息后，是否等待 broker 返回确认信息，0 代表无须等待，1 代表只需要 leader 返回，-1 代表所有复制节点完成同步后返回	0
request_timeout_ms	number	No	消息发送超时时间	10000
serializer_class	string	No	值的序列化处理类	" kafka.serializer.String Encoder "
topic_id	string	Yes	必填选项，输出消息的 topic 名称	
workers	number	No	工作线程数	1

## 12.2.8 filter 插件

filter 段中的插件完成对事件内容的修改，包括添加、修改、删除字段甚至直接将整条事件消息丢弃。本节将介绍 Logstash 中最常用的 grok 插件。

grok 插件是 Logstash 中最灵活、使用最多的将非结构化数据进行结构化的插件。一般的日志消息传入到 logstash 后，如果直接传递到 Elasticsearch 中，则整条日志内容将作为 message 字段内容存储。而为了将来在 Elasticsearch 全文搜索引擎中能够有效地进行查询，则应当将日志内容中的字段解析、分拆，形成多个独立字段存储。grok 能够帮我们实现此功能，它使用正则表达式来解析日志，获取日志信息。它可以将 Syslog、Apache、MySQL，以及其他常用的日志格式进行结构化存储，这些常规日志类型已有了现成的模板。

grok 通过文本正则表达式来匹配日志内容，其语法是 `%{SYNTAX:SEMANTIC}`。

SYNTAX 是要匹配到文本的模式名称，本质上是一个正则表达式，例如 3.44 这个值将用 NUMBER 这个模式匹配，而 NUMBER 在 grok 配置文件中是一个匹配数字的正则表达式。55.4.21.1 的值将用 IP 地址这个模式匹配，而 IP 地址在 grok 配置文件中是一个匹配 IP 地址的正则表达式。

SEMNATIC 可以认为是一个赋值变量名，当日志内容通过 SYNTAX 配合获取后，赋给 SEMNATIC 这个变量。例如，3.44 可能是一个事件的周期，那么我们将其变量名定义为 duration。而 55.4.21.1 是一个客户端 IP 地址，我们将其命名为 client。在解析完成且赋值成功后，在原有的 message 字段中多出了 duration 或者 client 字段，其值就是解析出来的对应内容。

通过上面的例子，我们将 grok 的表达式写成如下：

```
%{NUMBER:duration} %{IP:client}
```

在默认情况下保存的字段数据类型都是字符串型，如果需要注明特定类型，则在语法上需要追加一个后缀，例如我们希望将之前的 3.44 转换成整型，那么应该将其写成：

```
%{NUMBER:duration:int}
```

下面看一个完整的示例来理解 grok 的解析功能，假设有一条 HTTP 日志 log 信息：

```
55.4.21.1 GET /index.html 15824 0.043
```

如果不适用于 grok 进行数据结构化处理，则整条日志内容将作为一个 message 段输出，下面定义如下模式：

```
%{IP:client} %{WORD:method} %{URIPATHPARAM:request} %{NUMBER:bytes}  
%{NUMBER:duration}
```

下面的示例是一个完整的配置文件：

```
input {  
  file {  
    path => "/var/log/http.log"  
  }  
}  
filter {  
  grok {  
    match => { "message" => "%{IP:client} %{WORD:method} %{URIPATHPARAM:  
request} %{NUMBER:bytes} %{NUMBER:duration}" }  
  }  
}
```

经过 filter 的过滤处理之后，在日志内容中将多出以下字段。

- client: 55.4.21.1
- method: GET
- request: /index.html
- bytes: 15824
- duration: 0.043

我们看到上面提到的 IP、WORD、NUMBER 等都是在 Logstash 中提前定义好了的正则

表达式。除了使用这些自带的表达式，用户可以通过以下两种方法自定义。

第一种是直接将正则表达式写在配置文件中，其表达式为：

```
(?<field_name>the pattern here)
```

尖括号内为获取后的字段名，后面紧跟着正则表达式，之后以“？”开头，外面以括号分组。例如 postfix 日志的 queue id 字段是一个由 10 或者 11 个十六进制字符表示的值，我们要获取这段信息则可以直接在 Logstash 的 grok 配置文件中写入：

```
(?<queue_id>[0-9A-F]{10,11})
```

另外一种选择是将客户化的模式匹配格式存入指定文件中。例如将 postfix queue id 的模式匹配写入 ./patterns/postfix 文件中，其内容如下：

```
# contents of ./patterns/postfix:
POSTFIX_QUEUEID [0-9A-F]{10,11}
```

下面是一条 postfix 日志文件示例：

```
Jan 1 06:25:43 mailserver14 postfix/cleanup[21403]: BEF25A72965: message-id=20130101142543.5828399CCAF@mailserver14.example.com
```

filter 内容是：

```
filter {
  grok {
    patterns_dir => "./patterns"
    match => { "message" => "%{SYSLOGBASE} %{POSTFIX_QUEUEID:queue_id}:" }
  }
}
```

通过上面的匹配，将获取以下新的字段：

- timestamp: Jan 1 06:25:43
- logsource: mailserver14
- program: postfix/cleanup
- pid: 21403
- queue\_id: BEF25A72965
- syslog\_message: message-id=<20130101142543.5828399CCAF@mailserver14.example.com>

其中 timestamp、logsource、program 和 pid 是通过 SYSLOGBASE 模式获取的，queue\_id 是通过我们自定义的 POSTFIX\_QUEUEID 获取的。Logstash 中默认包含了大量的模式匹配让用户使用，这样就无须再进行用正则表达式编辑，可参考：<https://github.com/logstash-plugins/logstash-patterns-core/tree/master/patterns>。



### 12.2.9 codec插件

前面已经简单地介绍过了 `codec`，它的作用是改变事件的表述方式，它能够在 `input`、`filter`、`output` 各个段中使用，本节就 `multiline` 进行介绍。

日志消息并不是一行一行的，最典型的例子是 Java 异常堆栈的信息。`multiline` 插件的作用是从文件或者远端读取事件时，能够通过某些规则识别多行消息内容。

下面是一个 `multiline` 的配置格式：

```
input {
  stdin {
    codec => multiline {
      pattern => "pattern, a regexp"
      negate => "true" or "false"
      what => "previous" or "next"
    }
  }
}
```

在 `multiline` 中有三个选项，分别是 `pattern`、`negate` 和 `what`。`pattern` 是一个正则表达式，用来判断多行日志中的分隔符。`negate` 是取反的意思，默认为 `false`，也就是说当正则表达式匹配了，那么此内容为一条多行内容，继续进行 `what` 判断。`what` 的值说明了这条匹配的消息是与之之前的内容还是与接下来的内容合并到一行。

举一个简单的例子：

```
input {
  stdin {
    codec => multiline {
      pattern => "^\s"
      what => "previous"
    }
  }
}
```

这个 `codec` 配置是说任何以空白开头的行都应该被包含在前一段日志内容中。

然后看一个稍微复杂的示例：

```
file {
  type => "tomcat"
  path => [ "/var/log/tomcat6/catalina.out" ]
  codec => multiline {
    pattern => " (^\\d+\\serror)|(^.+Exception: .+)|(^\\s+at .+)|(^\\s+... \\d+
more)|(^\\s*Caused by: .+)"
    what => "previous"
  }
}
```

我们从文件中读取 Tomcat 的 Java 日志，其中用明确的正则表达式进行了分行判断，匹配上的日志行被包含到前一段日志内容中。

## 12.3 Elasticsearch

### 12.3.1 基本概念

#### 1. Lucene

在 PaaS 的日志集中管理中，我们使用 Elasticsearch 作为底层的搜索引擎。在大数据、搜索引擎平台流行之前，用户对于数据的存储与检索是通过关系型数据库实现的。最基础的方法是将文本内容存储在数据库表中，通过 SQL 语言的 like 操作符来匹配文档。不同的数据库在 like 操作符上又引入了高级功能，例如用正则表达式来支持更灵活的查询。即便引入了各类高级方法，用户还是有大量的工作要做，比如对查询输入的分词管理、设计在海量数据上的弹性可伸缩方案等。针对以上问题，开源社区开发了一个全文搜索库——Lucene，注意它是一个库文件，也就是说开发人员能够使用它来构建自己的全文搜索引擎，它为开发人员提供了全文索引与查询的具体实现。Elasticsearch 是基于 Lucene 库文件的搜索引擎，它是一个产品，开箱即用，提供丰富的接口来索引、检索数据。

Lucene 是 Elasticsearch 的基础，它是一个独立的 Java 库文件，具有高性能、可扩展、轻量级的具体实现。作为一个全文检索引擎，它具有如下突出的优点。

- 索引文件格式独立于应用平台。Lucene 定义了一套以 8 位字节为基础的索引文件格式，使得兼容系统或者不同平台的应用能够共享建立的索引文件。
- 在传统全文检索引擎的倒排索引的基础上实现了分块索引，能够针对新的文件建立小文件索引，提升索引速度。然后通过与原有索引的合并，达到优化的目的。
- 优秀的面向对象的系统架构，使得对于 Lucene 扩展的学习难度降低，方便扩充新功能。
- 设计了独立于语言和文件格式的文本分析接口，索引器通过接收 Token 流完成索引文件的建立，用户扩展新的语言和文件格式，只需要实现文本分析的接口。
- 已经默认实现了一套强大的查询引擎，用户无须自己编写代码即可使系统获得强大的查询能力，在 Lucene 的查询实现中默认实现了布尔操作、模糊查询、分组查询等。

图 12-10 体现了应用程序与 Lucene 的关系，开发人员在应用程序中可通过 Lucene 实现索引与查询功能。Lucene 采用的是一种被称为反向索引（Inverted Index）的机制。反向索引

就是说我们维护了一个词/短语表，则对于这个表中的每个词/短语，都有一个链表来描述有哪些文档包含了这个词/短语。这样用户在输入查询条件时，就能非常快地得到搜索结果。对文档建立好索引后，就可以在这些索引上面进行搜索了。搜索引擎首先会对搜索的关键词进行解析，然后在建立好的索引上面进行查找，最终返回和用户输入的关键词相关联的文档。

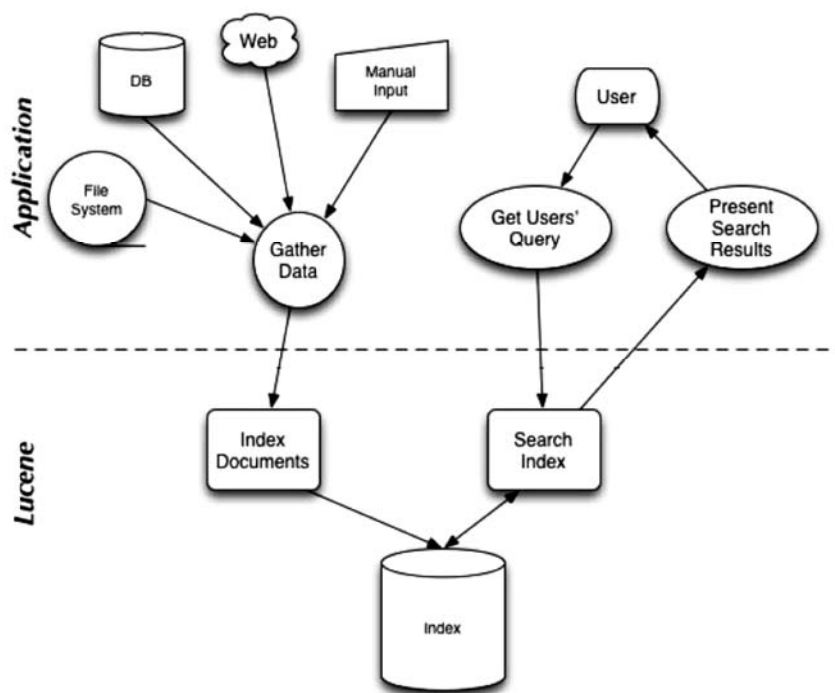


图 12-10 应用程序与 Lucene 的关系

## 2. Elasticsearch

Elasticsearch 是基于 Lucene 库构建的开源搜索引擎平台，它用 Java 语言实现，以 RESTful 的 Web 接口对外提供服务，交互内容采用 JSON 文档格式，其作者谢伊巴农（Shay Banon）于 2010 年将其开源贡献给了社区，之后 Elasticsearch 快速发展，目前和 Solr（另一个流行搜索引擎平台）并驾齐驱，成为互联网上最流行的两大企业搜索引擎平台。也许你对 Lucene 和 Elasticsearch 都有了一定的了解，但我们还是从介绍一些基本概念来认识 Elasticsearch。

### 1) 数据结构概念

Elasticsearch 是这样定义数据结构的。

#### ● Index

在 Elasticsearch 中，“Index”是一个名词，它将数据存放在一个或者多个 Index 中。Index 这个词很容易产生歧义，因为在计算机中它代表索引这个动作或者代表数据的索引。如果将其与关系型数据库进行类比，则实际上 Index 相当于之前数据库中的一张表，用来存

储用户的相关数据。

- Document

Document 是从 Lucene 中引入的一个概念，它相当于关系型数据库的一行记录。在 Lucene 搜索世界中，所有数据对象都是以文档方式存在的。在数据的结构化要求上，文档型（Document）要比行（Row）松散很多。在 Document 中也包含了很多字段（Field），这与关系型数据库行中的字段是一样的，但 Document 没有严格的规定一个文档有哪些字段及字段类型是什么，因此用户在将一个文档存储到一个 Index 时，可以灵活地增加、减少相关字段。Document 遵循 JSON 格式，结合 Logstash 来看，最终从 Logstash 中输出的就是一个包含多字段的 Document。

- Mapping

Mapping 是 Elasticsearch 中很重要的一个概念，刚刚说过在 Elasticsearch 中并没有类似于关系型数据库中的 Schema 概念，我们并不需要定义在一个 Document 中要有哪些字段。但是当 Document 发送过来并进行存储前，Elasticsearch 需要首先对文档中的字段进行索引，我们依据什么规则来进行索引呢？我们将 Mapping 理解为对字段进行索引的规则。一个 Mapping 由一个或多个 Analyzer 组成，一个 Analyzer 又由一个或多个 Filter 组成。Elasticsearch 在索引文档时，把字段中的内容传递给相应的 Analyzer，Analyzer 再传递给各自的 Filter。这些 Filter 完成最终的索引工作，包括分词、大小写忽略及删除标签等。Document 中的字段默认为对应的 Mapping，Elasticsearch 依据字段的类型来分配默认的 Mapping，当然用户可以自定义 Mapping，并在对文档进行索引前指定字段的 Mapping。

### 2) 组件概念

- Node 和 Cluster

Elasticsearch 以单实例形式运行 Node。单实例部署的 Elasticsearch 的 Node 可以满足一些基础场景的需求，但在生产环境涉及可用性、容错性、高性能等要求时，会将多个 Node 组合成一个 Cluster 集群。

- Shard

在集群部署的 Elasticsearch 多节点中，为了提升整体的性能，Elasticsearch 会将数据存储到多个节点的 Lucene Index 中，只有将这些节点中的数据聚合在一起才是一个完整的 Index，每个节点上的数据被称为 Shard，代表 Index 的一段分片。在用户向 Cluster 提交一次查询动作时，Elasticsearch 内部会自动完成各个节点之间的交互，从每个 Shard 中检索数据并完成聚合操作。

- Replica

Shard 用于将一份全量数据分成多片，放到不同的集群节点来提升检索性能。Replica 是

一份 Shard 的复制，它的作用是对同一份数据提供冗余，防止在某节点失效时数据丢失；同时 Replica 的份数越多，读取操作的性能也会有很大的提升。

### ● Gateway

Elasticsearch 运行时会产生大量的关于集群状态、Index 设置相关的元数据信息，这些数据在 Gateway 中持久化。

### 3) 索引与查询

搜索引擎平台提供的两个主要操作接口是索引、查询，在 Elasticsearch 集群中由于 Index 数据分散到各个节点中的 Shard 中，并且每一个 Shard 还有自己的冗余备份 Replica，因此每一次索引与查询动作都涉及 Cluster 集群中的多节点交互。

Elasticsearch 的索引接口有多种方式，我们可以使用 HTTP 发起索引访问，将文档发送给 Elasticsearch，也可以采用无状态的 UDP，牺牲可靠性来提供请求性能。一个请求索引的文档有一个或者自动生成一个 Document ID，Elasticsearch 依据此 ID 来判断该文档应该被存储在哪个 Shard 中，只有 Shard 所在的节点才能完成索引动作，Replica 节点仅仅同步数据和提供查询。如图 12-11 所示是一个索引操作流程，客户端发起一个文档索引请求，它可以连接到 Cluster 集群中的任意一个节点。在该例中连接的是 node\_2，node\_2 对 Document ID 进行判断，发现数据应当由 node\_1 上的 shard\_2 来处理，于是它将请求转发到 node\_1。

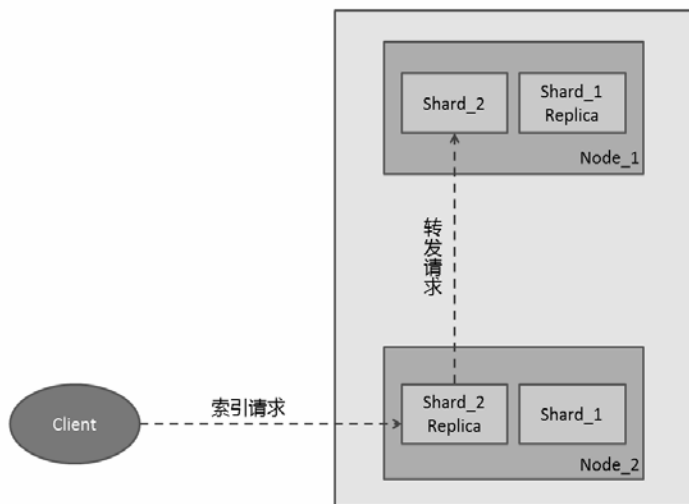


图 12-11 索引操作流程图

Elasticsearch 的检索请求要比索引更加复杂，在一个 Cluster 集群环境中，Elasticsearch 必须询问所有的 Node，将获取的数据汇总后才发送给用户。如图 12-12 所示是一次数据检索操作，客户端发起请求给 node\_2，node\_2 内部会对用户发起的检索请求进行处理，搜索相关数据，同时会与集群内的其他节点进行通信，收集在其他节点检索出来的数据，最

后，所有节点将数据发送到 node\_2 中进行汇总，统一返回给客户端。

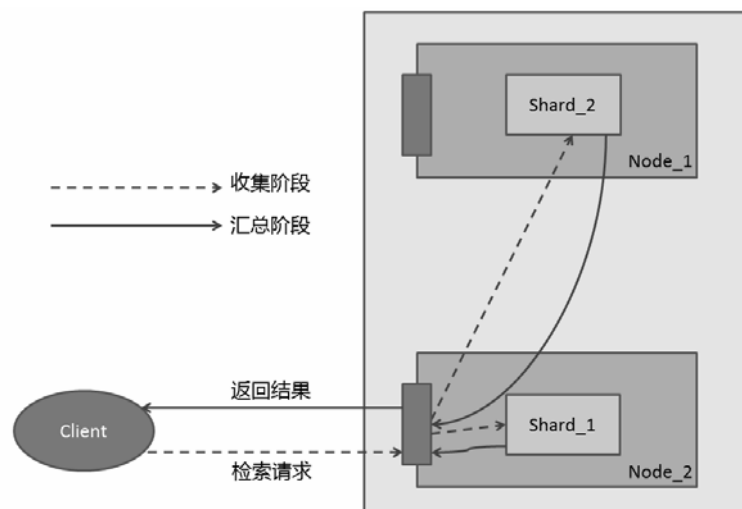


图 12-12 数据检索操作

## 12.3.2 安装与使用

### 1. 安装 JDK

Elasticsearch 要求至少是 Java 7 的运行环境，因此在安装 Elasticsearch 之前需要先安装 JDK 环境，当前的 Elasticsearch 版本是 1.7.0，官网推荐的 Oracle JDK 版本是 1.8.0\_25。

Linux 操作系统可以直接在 Oracle 网站下载 JDK 8，其文件名格式是 `jdk-8uversion-linux-x64.tar.gz`。将文件复制到 Java 安装目录，解压压缩包，将路径添加到 `JAVA_HOME` 环境变量中，通过下面的命令查看 JDK 版本：

```
java -version
echo $JAVA_HOME
```

### 2. 安装 Elasticsearch

下载 Elasticsearch 安装包，它支持 TAR、ZIP 安装包，也支持 DEB、RPM 等 Linux 安装包，为了简单起见，我们在 Linux 上使用 TAR 安装包，整个包有 27MB：

```
curl -L -O https://download.elastic.co/elasticsearch/elasticsearch/
elasticsearch-1.7.0.tar.gz
```

将下载的压缩文件复制到安装目录，解压压缩包：

```
tar -xvf elasticsearch-1.7.0.tar.gz
```

进入 bin 目录：

```
cd elasticsearch-1.7.0/bin
```

我们以单例模式运行 Elasticsearch

```
./elasticsearch
```

如果以上步骤都比较顺利，则会在控制台上看到如下消息：

```
[2015-07-21 09:01:10,395][INFO ][node                               ] [Blue Streak]
version[1.7.0], pid[297048], build[929b973/2015-07-16T14:31:07Z]
[2015-07-21 09:01:10,396][INFO ][node                               ] [Blue Streak]
initializing ...
[2015-07-21 09:01:10,522][INFO ][plugins                               ] [Blue Streak]
loaded [], sites []
[2015-07-21 09:01:10,592][INFO ][env                                   ] [Blue Streak]
using [1] data paths, mounts [[/ (/dev/mapper/VolGroup00-LVroot)]], net
usable_space [3gb], net total_space [7.7gb], types [ext3]
[2015-07-21 09:01:13,771][INFO ][node                               ] [Blue Streak]
initialized
[2015-07-21 09:01:13,771][INFO ][node                               ] [Blue Streak]
starting ...
[2015-07-21 09:01:13,975][INFO ][transport                             ] [Blue Streak]
bound_address {inet[/0:0:0:0:0:0:0:0:9300]}, publish_address {inet[/169.
254.95.120:9300]}
[2015-07-21 09:01:14,036][INFO ][discovery                             ] [Blue Streak]
elasticsearch/qgnPzZZqRC2WfnAGp4SwRw
[2015-07-21 09:01:17,819][INFO ][cluster.service                       ] [Blue Streak]
new_master [Blue
Streak][qgnPzZZqRC2WfnAGp4SwRw][cnsH231147][inet[/169.254.95.120:9300]],
reason: zen-disco-join (elected_as_master)
[2015-07-21 09:01:17,908][INFO ][http                                   ] [Blue Streak]
bound_address {inet[/0:0:0:0:0:0:0:0:9200]}, publish_address
{inet[/169.254.95.120:9200]}
[2015-07-21 09:01:17,908][INFO ][gateway                               ] [Blue Streak]
recovered [0] indices into cluster_state
[2015-07-21 09:01:17,908][INFO ][node                               ] [Blue Streak]
started
```

启动之后可以看到自动生成了一个 Node，为名“Blue Streak”，由于只有一个节点，因此将自己选举为 Master。集群中 Master 的具体含义将在 12.3.3 节介绍。

Elasticsearch 除了从文件中读取配置，也支持从命令行上获取参数，在启动时加入下面的参数，可以定义集群与节点名称：

```
./elasticsearch --cluster.name my_cluster_name --node.name my_node_name
```

### 12.3.3 REST API

在服务器启动之后，下一步是与 Elasticsearch 进行交互，默认情况下 Elasticsearch 通过 9300 端口提供 HTTP 的 REST API 接口，通过这些 API 接口可以完成以下任务：

- 检查 Cluster、Node 及 Index 的监控状态、统计信息；
- 管理 Cluster、Node 及 Index 元数据；
- 执行 CRUD 和查询数据操作；
- 执行高级的查询操作，例如排序、过滤、汇聚、分页、脚本等。

### 1. Cluster Health

在后面的示例中将使用 Curl 工具模拟发送 HTTP 请求对 Elasticsearch Node 进行访问。假如我们在同一台机器上执行命令，则下面的 Restful API 用于获取集群健康情况的信息：

```
curl '10.31.64.11:9200/_cat/health?v'
```

其返回值为：

```
epoch      timestamp cluster      status node.total node.data shards
pri relo init unassign pending_tasks
1437468590 16:49:50 elasticsearch green          1          1      0
0      0      0          0          0
```

我们看到默认的集群名称 Elasticsearch 正在运行中，其状态是“green”，我们从健康检查中可以获取“green”、“yellow”、“red”三个值。其中，“green”代表所有功能全部正常；“yellow”代表所有数据访问正常，但某些 Replica 存在问题，需要重新分配；“red”代表某些 Shard 数据存在问题。需要注意的是，即便是“red”状态，Elasticsearch 仍然能够使用，只有在查询这些问题 Shard 数据时才会出现问题。

从上面的信息可以看到，集群中只有一个 Node，并且暂时没有任何 Shard 数据。集群默认的名称是 Elasticsearch。现在可以立即启动其他几个节点构成一个 Elasticsearch 集群。如果要将 Elasticsearch 节点在不同的 OS 上启动，则在多网卡操作系统上最好指定服务发现组播和 Node 节点之间的通信网卡。我们可以在配置文件 ./config/elasticsearch.yml 中加入以下两行：

```
discovery.zen.ping.multicast.address: 10.31.64.11
network.host: 10.31.64.11
```

假设 10.31.64.11 是操作系统上的 IP 地址，第一行指组播服务发现的网卡使用 10.31.64.11 网卡，第二行指 Node 节点之间的通信网卡使用 10.31.64.11 网卡。在多网卡环境中，如果不指定该配置，则可能会导致 Elasticsearch 之间无法通信。我们在其他服务器上启动 Elasticsearch 节点，如果使用组播通信服务发现的网卡，则这些服务器的 IP 地址应该在同一网段，例如 10.31.64.12、10.31.64.13 等。

接下来，使用 Node 查询命令检查集群中的节点信息：

```
curl '10.31.64.12:9200/_cat/nodes?v'
```

此时我们看到在集群中有两个节点，其中有一主节点、一备节点：

```
host      ip      heap.percent ram.percent load node.role master
```



```

name
  cnsh231147 10.31.64.11          5          56 18.12 d          *
node1
  CNSH231162 10.31.64.12          7          65 18.70 d          -
node2

```

## 2. 创建 index

通过下面的命令在 Elasticsearch 中建立一个 Customer 的 Index，在 Elasticsearch 中建立 Index 时使用了 HTTP 的 PUT 方法。在最后加入了 pretty，表示需要服务器返回完整、可读的 JSON 格式响应。在返回结果中看到服务器的“acknowledge”响应为 true。

```

curl -XPUT '10.31.64.11:9200/customer?pretty'
{
  "acknowledged" : true
}

```

接下来我们检查 Elasticsearch 中的 Index 情况，我们看到其中有一个 Customer 的 Index，其 Shard 默认为 5，每份 Shard 有 1 份 Replica，因为在 Cluster 中有两个 Node，因此满足 Replica 为 1 份的要求，健康状态是 green：

```

curl '10.31.64.12:9200/_cat/indices?v'
health status index      pri rep docs.count docs.deleted store.size pri.
store.size
green open   customer    5    1           0           0     970b     575b

```

## 3. 创建 document

通过 PUT 方法在 Customer Index 中创建一个 Document，它是一个简单的文档，只有一个 name 属性：

```

curl -XPUT '10.31.64.11:9200/customer/external/1?pretty' -d '
{
  "name" : "fishriver"
}'

```

返回内容如下，我们看到返回结果中包含了 Index 名称、文档类型、在创建文档时自动分配的 ID 及版本号。在创建一个 Document 时并不一定要求 Index 存在，但在不存在所属 index 时会自动创建。

```

{
  "_index" : "customer",
  "_type" : "external",
  "_id" : "1",
  "_version" : 1,
  "created" : true
}

```

我们用 GET 方法，依据 Document ID 在 Index 中查找刚刚创建的文档，在返回结果的 \_source 字段中包含了完整的文档内容。

```
curl -XGET '10.31.64.11:9200/customer/external/1?pretty'
```

```
{
  "_index" : "customer",
  "_type" : "external",
  "_id" : "1",
  "_version" : 1,
  "found" : true,
  "_source" :
  {
    "name" : "fishriver"
  }
}
```

### 4. 修改 document

文档的修改有两种方式，一种是直接通过创建的 API 替换全部文档，一种是采用 `update` 方式修改。Elasticsearch 的数据存储方式是不可修改的，因此所有的修改都是以新增删除动作、新增文档、删除旧文档的方式实现对数据的修改。API 创建与 `update` 的区别如下。

- API 创建：新增 new 文档，替换索引，删除 old 文档。
- `update`：查找 old 文档，修改内容，新增 new 文档，替换索引，删除 old 文档。

下面的命令行将之前的 Customer 中 ID 为 1 的文档内容进行了替换：

```
curl -XPUT '10.31.64.11:9200/customer/external/1?pretty' -d '{
  "name" : "yuhe002"
}'
```

返回结果如下，其中的 `created` 字段为 `false`，代表文档 ID 已经存在，因此在这里进行了文档替换，此时老版本的文档并没有被立即删除：

```
{
  "_index" : "customer",
  "_type" : "external",
  "_id" : "1",
  "_version" : 5,
  "created" : false
}
```

下面我们查找并检查文档的内容：

```
curl -XGET '10.31.64.11:9200/customer/external/1?pretty'
{
  "_index" : "customer",
  "_type" : "external",
  "_id" : "1",
  "_version" : 5,
  "found" : true,
```

```

    "_source" :
    {
      "name" : "yuhe002"
    }
  }
}

```

通过 `update` 命令在文档中添加新字段 `fond`:

```

curl -XPOST '10.31.64.11:9200/customer/external/1/_update' -d '
{
  "doc" : { "fond" : "basketball" }
}'

```

检查 `update` 的文档中已经有了新字段内容:

```

curl -XGET '10.31.64.11:9200/customer/external/1?pretty'
{
  "_index" : "customer",
  "_type" : "external",
  "_id" : "1",
  "_version" : 6,
  "found" : true,
  "_source" : { "name" : "yuhe002", "fond" : "basketball" }
}

```

## 5. 删除 document、index

下面的命令通过 ID 删除相关文档:

```

curl -XDELETE '10.31.64.11:9200/customer/external/2?pretty'

```

通过查询方式删除相关文档:

```

curl -XDELETE '10.31.64.11:9200/customer/external/_query?pretty' -d '
{
  "query" : { "match" : { "name" : "yuhe002" } }
}'

```

删除整个 Index:

```

curl -XDELETE '10.31.64.11:9200/customer?pretty'

```

在使用 ELK stack 时, 与 Elasticsearch 交互的大部分工作都在 Logstash 内部完成, 但我们需要对基本的 Elasticsearch 概念有所了解, 并具备在脱离 Logstash 的情况下直接通过命令行查找、检索数据的能力。

## 12.3.4 集群设置

在前面章节我们构建了两个节点的 Elasticsearch 集群, 本节将详细地讨论集群的相关配置。Elasticsearch 集群中的节点类型有三类: Master 节点、Data 节点、Forward 节点。一个集群节点在启动后首先搜索与发现集群内是否存在 Master 节点, 如果存在则申请加入, 如

果不存在将进行选举，推举出一个 Master，这与 ZooKeeper 类似。Master 负责检查集群中其他节点的健康状况，确认它们能够响应请求。Master 同时承担接收新节点加入集群的请求处理的职责。在 Master 失效或者故障异常后，集群中的其他节点将发起一次选举，推举出一个新的 Master。

Data 节点是实际存储数据的节点，Elasticsearch 的检索、查询与索引功能都需要有 Data 节点的参与。一个节点可以集 Master、Data 职责于一身，在大集群中为了降低工作负载，常常将 Master 职责独立于 Data 之外。

当一个节点既不作为 Master 存在，又不存储数据时，它就是一个 Forward 转发节点，在加入集群后对内将发现集群中所有节点的分布，对外将接收检索、查询请求并将其转发给集群内真正的处理节点。Forward 可以用于实现一个简捷的负载均衡器，搭建一个单独的 Forward 节点给专门的应用使用。

Elasticsearch 允许一个节点承担多种职责，如果仅需让该节点以 Data 角色加入集群，则在 Elasticsearch.yml 配置文件中设置：

```
node.master: false
node.data: true
```

如果需要一个节点作为 Master 角色加入集群，则在 elasticsearch.yml 配置文件中做如下设置：

```
node.master: true
node.data: false
```

需要注意的是在一个集群中最终通过选举承担 Master 角色的只有一个节点，如果该节点在选举时没有被推举为 Master，同时它的 Data 角色也是关闭的，那么它将成为 Forward 角色。如果将这两个选项全部设置为 false，那么节点角色将成为 Forward。

假设在一个集群中有 10 个节点，由于网络异常或者其他原因，集群被分割为两个独立的部分，这时其中无法与 Master 通信的部分将选举出新的 Master，集群一分为二，这种情况被称为脑裂（Brain Split），在 ZooKeeper 分布式场景中也会有同样的问题存在。在 Elasticsearch 中，我们可以要求参加选举 Master 节点数过半来避免。如果集群中有 10 个节点，那么至少存在 6 个节点的投票选举才能成功，我们通过 discovery.zen.minimum\_master\_nodes 来配置这个选项。

在 Elasticsearch 配置文件中有一个 cluster.name 属性来设置集群名称，其默认值为 elasticsearch。Elasticsearch 默认采用网络二层组播方式发现其他节点，为了防止集群节点的误加入，在生产环境中一定要对 Cluster 名称进行设置。

### 1. 网络相关

Elasticsearch 集群默认采用二层组播的方式进行节点之间的相互发现，有四个相关的配置。

- `discovery.zen.ping.multicast.group`: 用来进行通信的二层组播 IP 地址, 默认为 224.2.2.4。
- `discovery.zen.ping.multicast.port`: 用来进行组播通信的端口, 默认为 54328。
- `discovery.zen.ping.multicast.ttl`: 组播请求的超时时间, 默认为 3 秒。
- `discovery.zen.ping.multicast.address`: 在多网卡的操作系统环境中, 可以选择一个网卡作为组播通信网卡, 将此选项配置为当前网卡所绑定的 IP 地址, 则表示使用此网卡进行组播通信。在选定好组播通信网卡后, 建议配置此选项。
- `discovery.zen.ping.multicast.enabled`: 如果想采用单播方式进行通信, 则将此选项设置为 `false`, 将 `multicast` 多播方式关闭后, 即可配置单播通信。
- `discovery.zen.ping.unicast.hosts`: 指定单播通信的集群中服务器地址。

该选项可指定主机、端口的列表信息, 例如:

```
10.34.12.1:9300, 10.34.12.2:9300, 10.34.12.3:9300
```

端口设置可采用 `range` 方式, 指定一个端口范围:

```
10.34.12.1:[9300-9399], 10.34.12.2:[9300-9399], 10.34.12.3:[9300-9399]
```

在集群中的节点完成发现工作后, Master 与其他节点会进行 `ping` 探测来检查对方的运行状态。伴随着 `ping` 动作的有配置频率、超时时间、重试次数。

- `discovery.zen.fd.ping_interval`: 定义节点之间相互 `ping` 的频率, 默认为 1s
- `discovery.zen.fd.ping_timeout`: 在每次发出 `ping` 数据包后, 会等待对方的回复, 如果在 `ping_timeout` 时间内没有收到回包, 则认为对方异常, 其默认时间为 30 秒。
- `discovery.zen.fd.ping_retries`: 在没有收到回包后重试发起 `ping` 的探测次数, 默认为 3。

## 2. 存储相关

Elasticsearch 的 `store` 模块定义了采用什么方式来存储数据。从操作系统层面上来看, Elasticsearch 允许我们将数据持久化到磁盘上, 也可以采用内存来放置数据。在 `index.store.type` 配置选项中, 有 4 种方式来存放数据, 具体配置如下。

- `simplefs`: 简单的文件系统, 它是一种基于磁盘的存储方式, 通过随机访问方式查询 Index 数据文件。在并发访问上性能欠佳, 在生产环境下不建议使用此方式。
- `niofs`: 这也是一种基于磁盘存储的方式, 它使用了 Java NIO 类来非阻塞地访问 Index 数据文件。在并发访问的场景下能有一个比较好的性能, 但由于 Java 实现在 Windows 平台上的相关 Bug, 建议在 Windows 平台上不要使用。
- `mmapfs`: 这是一种基于内存映射的磁盘存储方式。它将磁盘文件的地址空间直接

映射到内存地址，减少了地址转换对资源的消耗。在 64 位操作系统上由于进程地址空间的大大增加，采用此方式会有一个非常客观的性能。

- **memory**: 用于直接将 **index** 数据文件放入内存中，该类型需要有足够的物理内存来存放数据。这种存储方式可以获得一个非常好的性能，但由于没有将数据持久化到磁盘中，所以机器故障时会造成数据丢失，对性能要求苛刻，但在不要求数据全量的情况下，可以采用这种存储方式。

从性能改善层面上看，4 种存储类型让我们有了更多的选择。我们还可以从硬件层面来做进一步的提升，例如采用更多的 SSD 固态硬盘作为磁盘存储。

### 3. 内存相关

Elasticsearch 作为搜索引擎平台，涉及大量数据的读写交互，提供足够的内存资源能够有效提升处理性能。建议 JVM 采用 64 位虚拟机，这样 Elasticsearch 节点进程内存可以设置为超过 4GB，Elasticsearch JVM 内存设置建议在 32GB 内。

读写缓存数据的内存参数指标一般有两个：内存大小与刷新频率。内存越多，在性能上获得的提升越大。刷新频率越长，性能越好，但数据的实时性无法得到保证。

#### 1) field data cache

每进行一次数据查询都会从各个节点的磁盘中加载大量的数据，数据发送之前的排序、汇聚处理将放在 **field data cache** 中进行。构建一次 **field data cache** 的代价很高。

- **indices fielddata.cache.size**: 用来设置 **field data cache** 的总内存大小，可以直接设置空间大小，例如 2GB，也可以设置为 Elasticsearch 总体 **heapsize** 大小的百分比，例如 20%。其默认值是 **unbounded**，即不做限制。
- **indices fielddata.cache.expire**: **field** 的超时时间，也是重新刷新 **field cache** 的时间，默认值为 -1，不做刷新动作，该选项尚在试验阶段，建议保留默认设置。

#### 2) circuit breaker

在 Elasticsearch 的内存中缓存了大量的数据，同时要处理大量的数据查询请求，从磁盘中加载不同的数据，从而使内存使用率的波动非常大，极易出现内存溢出的状况。在 Elasticsearch 中引入了“断路”的保护机制，在加载数据到内存的过程中，它会判断当前内存与整体 JVM 的比例，如果超出阈值，则停止加载，先进行内存回收。

- **indices.breaker.total.limit**: **circuit breaker** 的最高级别设定，默认值是 JVM **heap** 的 70%，在内存达到这个数量时会触发内存回收。
- **indices.breaker.fielddata.limit**: 当系统发现 **fielddata** 的数量达到一定数量时会触发内存回收，默认值是 JVM **heap** 的 70%

- `indices.breaker fielddata.overhead`: 系统是要加载 `fielddata` 时会进行预先估计,当系统发现要加载进内存的值超过 `limit*overhead`, 则会进行内存回收, 默认是 1.03。
- `indices.breaker.request.limit`: 这种断路器是 Elasticsearch 为了防止 OOM (内存溢出), 在每次请求数据时设定的一个固定的内存数量, 默认值是 40%。
- `indices.breaker.request.overhead`: 同上, 也是 Elasticsearch 在发送请求时设定的一个预估系数, 用来防止内存溢出, 默认值是 1。

### 3) Index Buffers

Buffer、Cache 都表示内存缓存。二者的区别在于: Buffer 是随后要写入磁盘的数据, Cache 则是从磁盘读出的以后会使用的数据; Buffer 用于提升写操作性能, 而 Cache 则用于提升读操作性能。Elasticsearch 在进行数据存储时, 并不会同步地将数据全部写入磁盘, 而是先缓存到内存中。`indices.memory.index_buffer _size` 用来指定我们可缓存的最大内存数量, 其默认值是 JVM 全部空间的 10%。`index.refresh_interval` 定义 buffer 缓存的刷新频率, 默认值为 1 秒。

## 4. 线程相关

线程设置用于提升处理计算能力。在单进程条件下并不是工作线程数越多, 性能越好, 还依赖于 I/O 模型、操作系统环境等。例如所有线程都会等待在一个共享的 I/O 资源上, 线程越多反而造成更多的操作系统上下文切换, 导致性能更加缓慢。在 I/O 模型一致、线程间无共享数据、操作系统计算机资源满足的条件下, 可以认为线程越多, 计算能力越强。在线程设置上一般有两个常规设置: 线程数和等待队列数。线程数代表在进程中可以用于计算的线程个数, 而等待队列数代表当工作线程都处于繁忙状态时运行的请求排队数。在 Elasticsearch 中对线程设置也有这两个指标, 不同的是 Elasticsearch 按照不同的功能将线程分为了不同的线程池。

Elasticsearch 的线程池类型有两种: `cached`、`fixed`。`cache` 线程池是一个无限大的线程池, 如果有很多请求, 则会创建很多线程。`fixed` 线程池保持固定个数的线程来处理请求, 默认设置与 CPU 核数相关, 其需要设置一个等待队列, 当一个请求到来但队列满了时, 会直接拒绝该请求。

Elasticsearch 节点会依据不同的功能划分不同线程池, 下面列出比较重要的四个。

- 索引 (`index`): 主要是索引数据和删除数据操作, 默认为 `fixed`, 线程数与 CPU 核数一致, 等待队列为 300。
- 搜索 (`search`): 主要是获取, 统计和搜索操作, 默认为 `fixed`, 线程数为 CPU 核数的三倍, 等待队列为 1000。
- 批量操作 (`bulk`): 主要是对索引的批量操作, 默认为 `fixed`, 线程数与 CPU 数量一

致，等待队列为 50。

- 更新 (refresh)：主要是更新操作，默认是 cached 类型。

例如我们要配置 index 功能的线程池，下面将配置其类型为 fixed，线程数为 100，等待队列为 500：

```
threadpool.index.type: fixed
threadpool.index.size: 100
threadpool.index.queue_size: 500
```

另外，线程池的配置调整是动态的，通过 API 可以更新线程配置。下面是更新操作：

```
curl -XPUT 'localhost:9200/_cluster/settings' -d '{
  "transient" : {
    "threadpool.index.type" : "fixed",
    "threadpool.index.size" : 100,
    "threadpool.index.queue_size" : 500
  }
}'
```

### 12.3.5 备份恢复

Elasticsearch 提供了一种非常简单数据备份与恢复方案，它让用户通过 API 将数据备份到远程存储上，这个存储可以是共享文件系统（例如 NAS）、Amazon S3、HDFS 和 Azure Cloud。

Elasticsearch 的备份过程是：先建立一个仓库，在仓库中可以存放一个或多个快照，在快照中备份具体的索引，一个快照可以备份一个或多个索引。

下面的 API 建立了一个名为 my\_backup 的仓库，其存储类型为 fs，即共享文件系统，存储位置为 /mount/backups/my\_backup：

```
curl -XPUT http://localhost:9200/_snapshot/my_backup
{
  "my_backup": {
    "type": "fs",
    "settings": {
      "compress": "true",
      "location": "/mount/backups/my_backup"
    }
  }
}
```

随后创建一个名为 snapshot\_1 的快照，同时备份 index\_1、index\_2 数据文件，如果没有 JSON 文档，则表示备份所有 index：

```
curl -XPUT http://localhost:9200/_snapshot/my_backup/snapshot_1
{
```



```
"indices" : "index_1,index_2",
}
```

通过下面的命令删除快照：

```
curl -XDELETE http://localhost:9200/_snapshot/my_backup/snapshot_1
```

通过下面的信息查看快照信息：

```
curl -XGET http://localhost:9200/_snapshot/my_backup/snapshot_1
```

Elasticsearch 的数据恢复非常检查，下面表示恢复所有索引数据：

```
curl -XPOST http://localhost:9200/_snapshot/my_backup/snapshot_1/_restore
```

如果我们需要恢复具体的 index，则加入下面的 JSON 文档内容：

```
curl -XPOST http://localhost:9200/_snapshot/my_backup/ snapshot_1/_restore
{
  "indices" : "index_1,index_2",
  "ignore_unavailable" : "true",
  "include_global_state" : false,
  "rename_pattern" : "index_(.+)",
  "rename_replacement" : "restored_index_$1"
}
```

### 12.3.6 监控管理

项目重新构建了一套系统、流程、工具，而运维则保证了系统的稳固运行，监控是保证稳定运行的重要手段。最后让我们回到监控层面，看看 Elasticsearch 的主要监控方法。我们可以使用多种协议来进行监控，JMX 检查 JVM 的运行情况，HTTP 检查应用的逻辑状态。监控要检查的信息包括是否存活、组件是否正常、性能情况如何。存活性用简单的 HTTP 健康检查实现，在组件层面上需要查看 Cluster、Node、Index 的情况，最后在性能方面要查看队列情况、内存使用等指标。

本节我们主要通过 Elasticsearch 提供的 HTTP Restful API 来检查各组件情况，因为 Elasticsearch 使用的是 JVM，所以基本的性能数据可以通过 JMX 获取。

Elasticsearch 的 HTTP 响应结果返回的是 JSON 格式的文档，这是计算机很好理解的、不会出现歧义的一种文档格式。但人们更需要直观的结果，特别是将结果直接返回到远程 SSH 终端。Elasticsearch 中的 cat API 满足了我们的这一需求，它将内容格式化成果格样式，便于查看。

在使用 cat API 之前先熟悉下其规则。cat API 的前缀是“hostname:9200/\_cat/”，之后是具体的查询对象，例如 health、nodes、indices、thread\_pool 等。

通过 health 来检查 Cluster 的状态，其中 status 代表节点的状态。下面的集群由于只有一

个节点，Shard 没有相应的 Replica，因此状态为 yellow:

```
$ curl localhost:9200/_cat/health
1438245782 16:43:02 elasticsearch yellow 1 1 26 26 0 0 26 0

$ curl 'localhost:9200/_cat/health?v'
epoch      timestamp cluster      status node.total node.data shards
pri relo init unassign pending_tasks
1438245810 16:43:30 elasticsearch yellow          1          1    26
26      0      0          26          0
```

每个对象的后面可接一个 **help** 参数来显示该对象有哪些可选列，例如我们在 **health** 后加入 **help**:

```
$ curl -s localhost:9200/_cat/health?help
epoch          | t,time                      | seconds
since 1970-01-01 00:00:00
timestamp      | ts,hms,hmmss               | time in
HH:MM:SS
cluster        | cl                           | cluster
name
status         | st                           | health
status
node.total     | nt,nodeTotal                | total
number of nodes
node.data      | nd,nodeData                 | number of
nodes that can store data
```

可以看到 **health** 对象有很多可选列使用，接着用 **h** 参数来定义我们需要的列，例如我们仅仅需要 **cluster**、**status**、**node.total**:

```
$ curl -s localhost:9200/_cat/health?h=cluster,status,node.total
elasticsearch yellow 1
```

接下来让我们查看 **indices**、**nodes** 信息:

```
$ curl -s localhost:9200/_cat/indices?v
health status index          pri rep docs.count docs.deleted
store.size pri.store.size
yellow open  accounts          5  1    1000          0
417.5kb    417.5kb
yellow open  logstash-2015.05.18 5  1     4631          0
17.2mb     17.2mb
yellow open  .kibana            1  1         3          0
13.7kb     13.7kb
yellow open  logstash-2015.05.20 5  1     4750          0
18.2mb     18.2mb
yellow open  logstash-2015.05.19 5  1     4624          0
17.7mb     17.7mb
yellow open  shakespeare        5  1    111396         0
18.1mb     18.1mb
```

```
$ curl -s localhost:9200/_cat/nodes?v
host          ip          heap.percent ram.percent  load node.role master
name
CNSZ031314 10.33.64.11          27          33 32.92 d          *
U-Go Girl
```

最后查看 thread\_pool 信息:

```
$ curl -s localhost:9200/_cat/thread_pool
CNAZ081314 10.38.64.11 0 0 0 0 0 0 0 0
```

thread\_pool 将线程池按功能划分为 bulk、index、search 三类，每一类都可以看到当前活动数、队列等待数，以及丢弃连接数。

## 12.4 Kibana

### 12.4.1 Kibana 介绍

Kibana 是基于 Elasticsearch 而设计的开源数据分析与可视化平台。在 PaaS 中的节点日志已经完成收集与索引后，我们使用 Kibana 对数据进行展示。Kibana 将服务器、业务逻辑、前台页面全部打包在一起，直接运行在 JVM 环境中，可快速完成安装启动。用户通过浏览器访问 Kibana，Kibana 调用 Elasticsearch API 来获取存储在 Elasticsearch 中的数据，之后在前台页面将数据可视化。

Kibana 的功能模块简单实用，并且一环扣一环，由 discover、visualize、dashboard 三个模块构成。其中 discover 用于数据检索查询，它支持 Lucene 查询语法、Elasticsearch 查询 DSL 两种检索方法。visualize 是紧挨着 discover 模块的，是数据可视化模块，使用 discover 中查询出来的数据，按照字段、字段类型等进行分组、排序、求和等一系列数学函数操作，最后形成图形报表输出。这些图形包括饼图、直方图、线形图等。

为了能够正确地使用 Kibana，我们必须保证 Elasticsearch 的版本为 1.4.4 及以上版本，本书中使用的 Kibana 版本为 4.1.1。

首先在官网 [www.elastic.co](http://www.elastic.co) 下载相应的安装包，之后解压安装包，进入安装目录：

```
# tar zxf kibana-4.1.1-linux-x86.tar.gz
# cd kibana-4.1.1-linux-x86
```

Kibana 的默认端口为 5601，我们可以修改 config/kibana.yml 配置文件，将 5601 端口号修改为习惯上的 80 端口。或者在前端采用负载均衡设备，分发到后端构建的多个 Kibana 节点上，前端负载均衡采用 80 端口，后端 Kibana 维持端口不变。

在 Kibana 配置文件中默认设置连接的 Elasticsearch 地址为 <http://localhost:9200>，基于实

实际情况的需要，在生产环境中为了保证数据的安全性，连接 Elasticsearch 可能需要配置用户名或者证书，相关配置如下：

```
# The Elasticsearch instance to use for all your queries.
elasticsearch_url: http://localhost:9200
kibana_elasticsearch_username: user
kibana_elasticsearch_password: pass

# If your Elasticsearch requires client certificate and key
kibana_elasticsearch_client_cert: /path/to/your/client.crt
kibana_elasticsearch_client_key: /path/to/your/client.key
```

在生产环境中，我们可以在创建 Kibana 实例的同时，在相同的主机上部署一个 Elasticsearch 节点，该节点加入到 Cluster 中作为 forward 角色使用，仅向主机 Kibana 提供服务，供集群内部负载均衡器使用。

最后启动 Kibana：

```
# ./bin/kibana
```

我们在浏览器中输入 Kibana 服务端的地址 `http://localhost`，页面显示如图 12-13 所示。

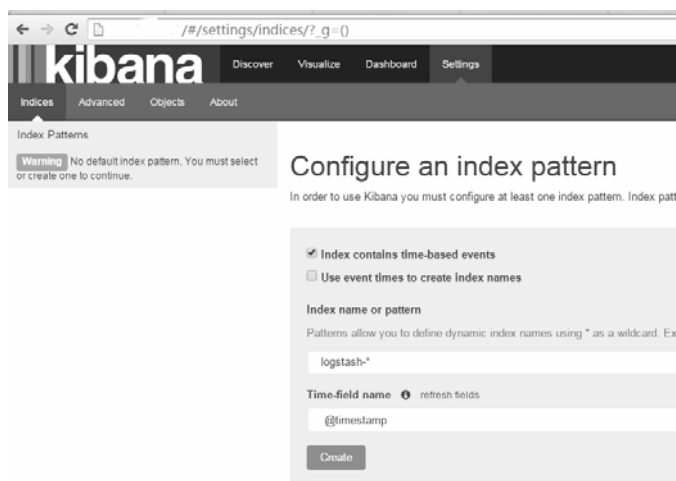


图 12-13 Kibana 主页

默认在没有添加 Index 的情况下，Kibana 会跳入 “Settings” 设置面板的 “Indices” 界面，在这里我们进行最基础的数据录入设置，在 “Index name or pattern” 中选择需要检索的 Elasticsearch Index，该栏支持模糊匹配模式，录入 “logstash-\*” 会将 Logstash 按日期保存的所有 Index 包含进来。另一个需要勾选的是 “Index contains time-based events”，该字段的含义是对于有 time 字段的 Index，在随后的 Discover 面板上将产生按时间轴排序的直方图。图 12-13 中选择的 Time 字段是 “@timestamp”。对于没有 time 字段的 Index，无法进行

最后的创建动作，需要取消 “Index name or pattern” 勾选。最后单击 “Create” 完成初始配置。

## 12.4.2 discover功能

整个 discover 面板非常简洁地分为了三大区域，每一个区域履行自己的职责，区域的划分如图 12-14 所示，区域 1 为检索区，区域 2 为字段区，区域 3 为展示区。

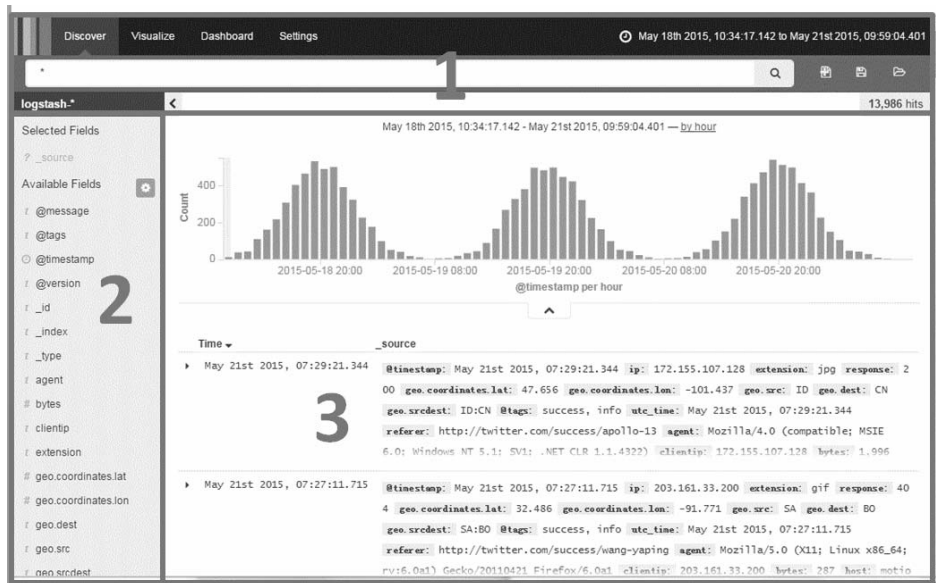


图 12-14 Kibana 功能界面

### 1. 检索区

我们在检索区的搜索栏中输入需要查找的内容后单击提交，其支持简单的字符串查询、Lucene 查询语法，以及基于 JSON 格式的 Elasticsearch 自定义查询语言。

在单击提交后，字段区域、展示区域的数据会同步更新显示，匹配的文档总数会在右上角显示。

最简单的查询方式是在输入框中直接输入文本字符串，Kibana 将从 Elasticsearch 中把所有符合匹配的文档检索出来并在展示区显示，默认为 500 行。

另外，我们还可以专门针对某一特别的字段进行检索，例如在 HTTP 的 access 日志中有响应状态码字段，假设该字段在 JSON 文档中的名称为 response，那么键入 response:200 将展示所有返回状态码为 200 的请求内容。对于字段搜索还可以指定查询范围，例如 response:[400 TO 499]，展示状态码为 400~499 的所有请求内容。Kibana 支持条件组合，通过 AND、OR、NOT 等操作符可将多个条件组合在一起，如图 12-15 所示为按照多个条件组

合进行搜索。



图 12-15 Kibana 搜索界面

在添加 index 时，如果我们选择了时间戳，那么在检索区的右上侧会出现一个时钟图标，单击它，我们看到有三个简单的时间范围查询方法。其中一个绝对查询，也就是在日历上选择开始、结束时间段，如图 12-16 所示。

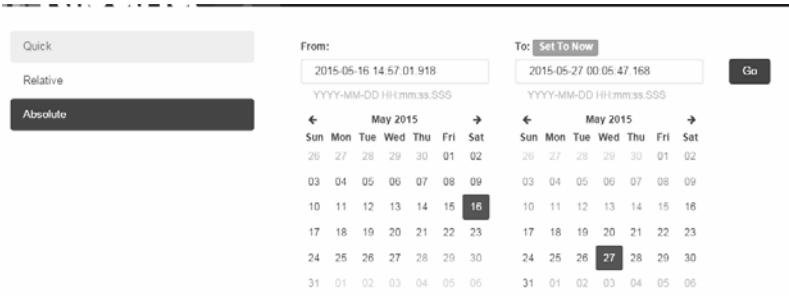


图 12-16 Kibana 日期选择界面

另外一个相对查询，以当前时间或者当月的第一天为基准，以某年某月某日某分等为条件进行时间检索，如图 12-17 所示。

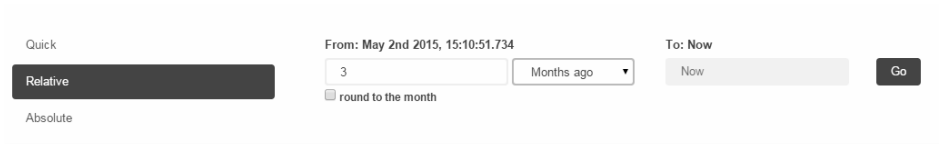


图 12-17 Kibana 日期相对查询界面

最后是快速查询方式，其定义了一些常用的时间检索条件，例如一年前、一个月前等，无须用户进行输入，如图 12-18 所示。



图 12-18 Kibana 快速查询日期界面

对于实时查看的数据，例如条件为 15 分钟以前，我们可以定义数据自动刷新，这样在面板上可以呈现动态更新的状态，点击“Auto-refresh”，我们可以打开自动刷新功能，并定义刷新频率，如图 12-19 所示。

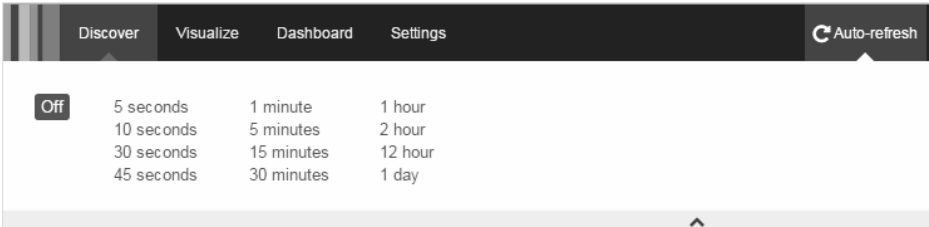


图 12-19 Kibana 自动刷新界面

对于每一次的搜索条件，我们都可将其保存，以供后续使用。通过单击搜索栏旁边的打开、保存图标来开启此功能，如图 12-20 所示。

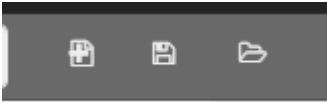


图 12-20 Kibana 的自动保存功能

## 2. 字段区

在字段区会默认将 index 中所有包含的字段列出来，我们可以单击设置图标从 Analyzed、Indexed、Type、Filed name 中过滤字段列表，如图 12-21 所示。

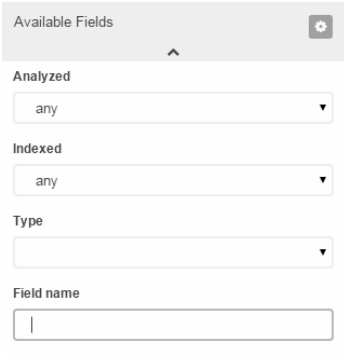


图 12-21 Kibana 可查字段区

将鼠标移动到某一字段上，将显示“add”按钮，单击它，就会把该字段加入展示区的数据表中。展示区的数据表默认显示时间戳字段及\_source 字段。图 12-22 将 agent 字段加入展示区中进行显示。

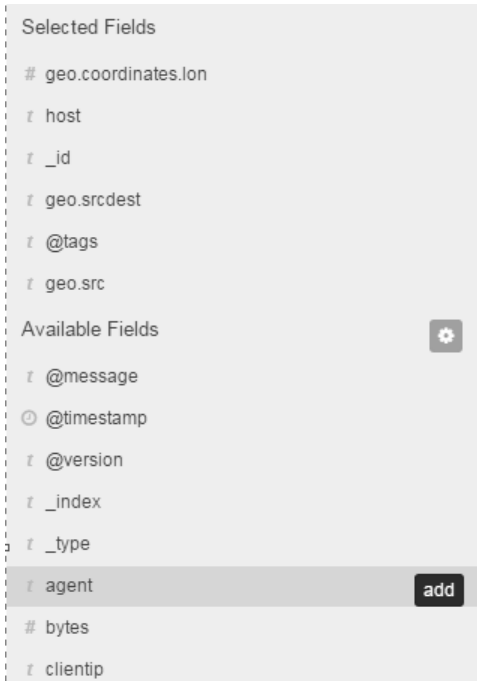


图 12-22 Kibana 字段显示区 1

单击其中一个字段，字段行会展开，该字段 TOP5 的计数统计如图 12-23 所示。我们单击了 request 字段，其展示了请求最多的 5 个 URL 列表。在列表的右边有两个放大镜，一个“+”号，一个“-”号，单击它后将以此字段作为查询条件（加号）或过滤条件（减号）进行检索。

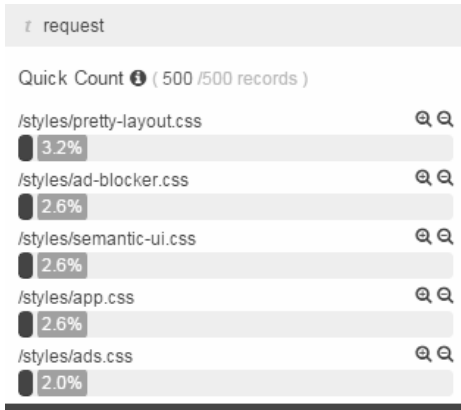


图 12-23 Kibana 字段显示区 2



单击后，这些条件会以标签的形式在检索区的搜索栏下显示，如图 12-24 所示。

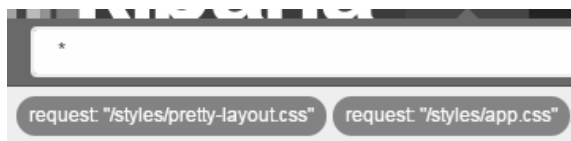


图 12-24 Kibana 字段显示区 3

鼠标滑动到标签时，会显示如图 12-25 所示的界面，分别代表取消该条件、在后续 visualize、dashboard 面板继续使用该条件、条件取反、条件删除。



图 12-25 Kibana 条件选择区

### 3. 展示区

最后让我们看一下展示区的功能，展示区默认按照时间排序，在字段区加入了新的字段后，我们可以通过单击数据表的字段头来以新字段排序。

单击数据表的任意一行，它将被展开，其中可以采用 Table 或者 JSON 的方式查询内容。如图 12-26 所示。

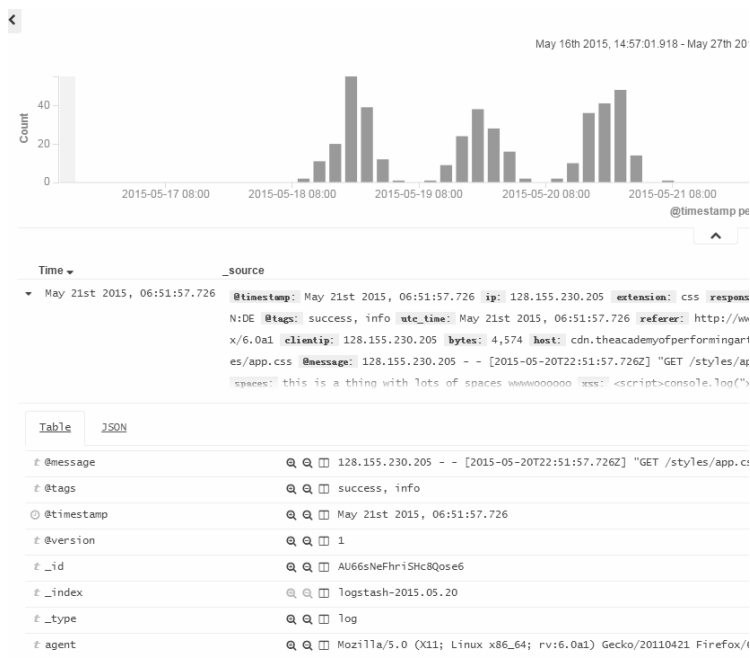


图 12-26 Kibana 日志展示区

在 Table 中有三个图标功能，它与字段区的添加条件、添加字段列功能一致。如图 12-27 所示。



图 12-27 Kibana Tables 功能图标

### 12.4.3 visualize 功能

visualize 是 discover 功能的延续，它将检索出来的数据以图形的方式进行展示。要快速地掌握 visualize 的使用方法，我们先要弄清楚几个概念。

visualize 是基于 Elasticsearch 的 aggregation 功能实现的。aggregation 就是将检索回来的数据进行一个汇聚处理，在 aggregation 中定义了两种类型的处理方法。Bucket 和 Metric。在 Kibana 的 visualize 功能里有很多 Bucket、Metric 选项。我们可以用一种更加简单的方式来理解这两种类型的汇聚处理。

Bucket 是“桶”的意思，我们可以形象地将这个动作理解为将数据按一定条件放到不同的桶中，是的，即分组。对于数值型字段，我们可以按照范围分组，例如将数字按 0~100、100~500、500 以上分为三组；对于字符串型字段，其直接按照内容分组。另外一个重要的分组类型是时间，虽然时间是连续的，但当我们把数据展现在一个图形中时，它会按照时间粒度进行分组。

Metric 是度量的意思，我们对聚集的文档进行函数计算，这些函数有求和、求总数、求平均值等。

依据上面对 Bucket、Metric 的理解，我们会发现它们与关系型数据库中 SQL 语言的分组和函数处理是类似的。在 visualize 定义图形时，一般的规则是对数据先进行分组，之后对每组的内容进行函数计算。对于 X、Y 轴的图形，函数计算的值放在 Y 轴，而分组对象放在 X 轴。一个简单的例子是按时间段展示的直方图，Y 轴是每个时间段的总数，而 X 轴是时间分段的连续。

更进一步来讲，对于生成复杂的图形，在分组之后的其中一个分组中，我们还可以继续以另一个字段作为新的分组条件来进行 Bucket 操作。在 visualize 中称为“sub-buckets”。

创建一个 visualize 数据图形，分为三个步骤，第一步选择图形类型，如图 12-28 所示，Kibana 支持区域图、数据表、线形图、饼状图等。

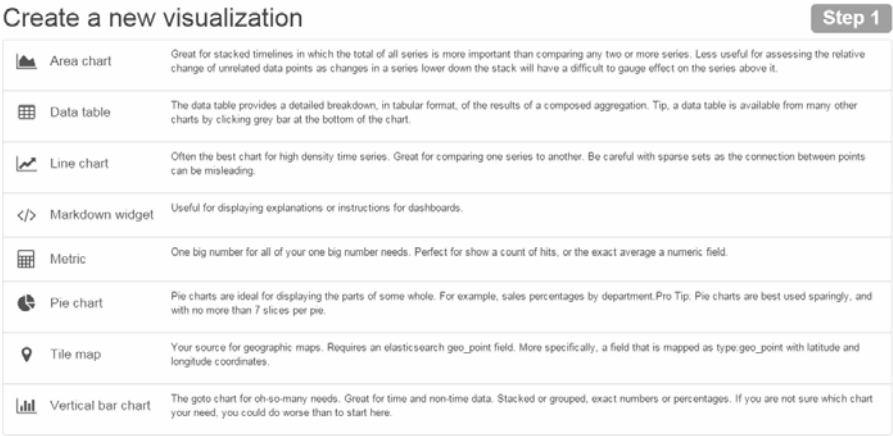


图 12-28 visualize 数据图形类型

第二步选择数据来源，我们可以选择新建一个查询，即重新选择索引，之后在检索栏输入查询条件。也可以选择在一个在 discover 面板中保存的查询。如图 12-29 所示，我们可以选择之前从 discover 中保存的名为“request”的查询。



图 12-29 选择检索源界面

最后一步进入了图形编辑页面，我们可以看到其界面风格与 discover 是一样的，如图 12-30 所示，它被分为三个区域：区域 1 为检索区；区域 2 为字段区；区域 3 为展示区。



图 12-30 图形编辑页面

区域 1 的检索区与在 discover 中是一样的，由于我们在第二步所选择的是一个保存了的查询“request”，因此搜索栏在图 12-30 中被屏蔽了。字段区则是对 metrics 及 buckets 的一些选择，之后在展示区展现图形。

在图 12-29 中创建了一个饼状图，在字段区的 metrics 中默认采用“count”来进行函数计算，之后我们选择 buckets，即分组方式，在选择时有两种类型的 buckets 图，一种是 Split Slices，它表示分组后在一个饼状图中显示；一种是 Split Chart，它表示在有 sub-bucket 子分组的情况下，数据在多个图形中展示。

让我们选择在“split slices”之后定义分组条件，将其设置为以“Terms”分组，Term 是术语的意思，在这里表示按照其中一个字段进行分组，我们选择“clientip”，并且仅展现 Top 8 的数据，按照 Count 计数进行排序。最后单击上方的“apply change”图标，生成图形。如图 12-31 所示是 buckets 条件图。

图 12-31 buckets 条件图

生成的饼状图如图 12-32 所示。

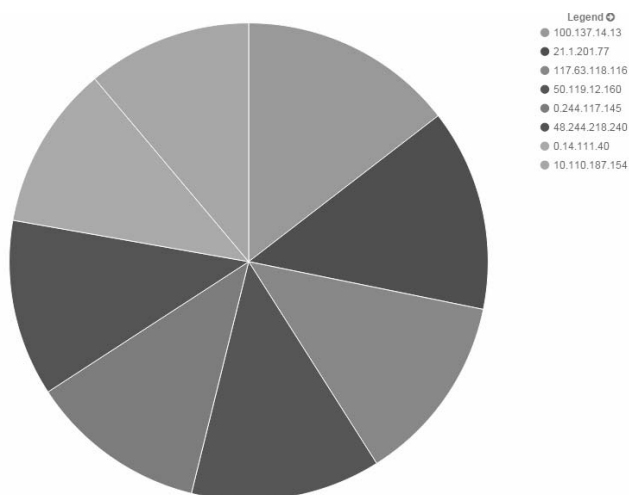


图 12-32 初次生成的饼状图

随后在图 12-31 所示的界面，再选择一个 sub-buckets 进行子分组，以 agent 为条件，如图 12-33 所示。

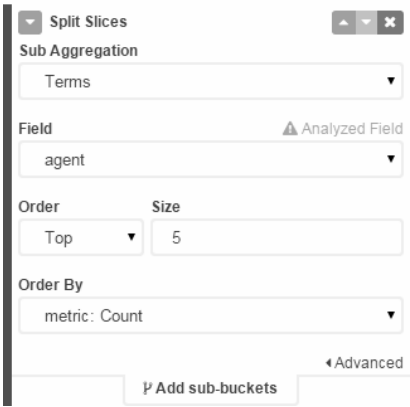


图 12-33 buckets 的二次选择

最后生成的图形如图 12-34 所示。

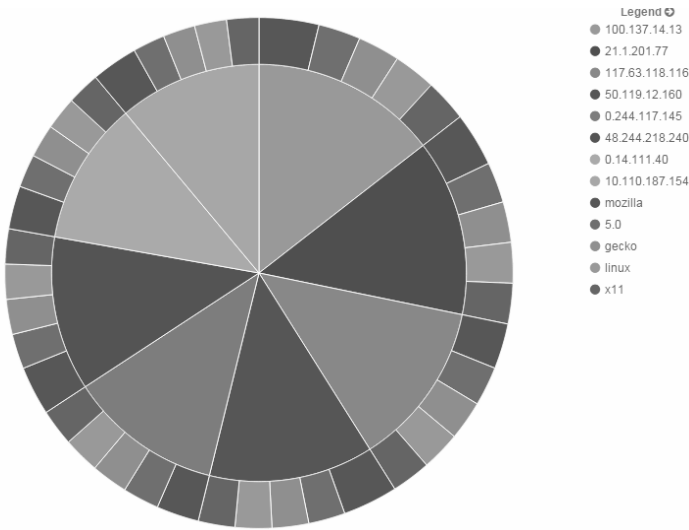


图 12-34 最后生成的饼状图

生成的图形可以保存下来作为后续的 Dashboard 使用，另外，区域 1 的检索区域同样可以依据时间来设置页面自动刷新，从而使图形动态化。

12.4.4 Dashboard功能

最后要介绍的是 Dashboard 功能，它是 visualize 的延续，如图 12-35 所示，在此界面选择了多个已经保存好的 visualize 图形，将用户关注的内容在一个面板中进行了展示。

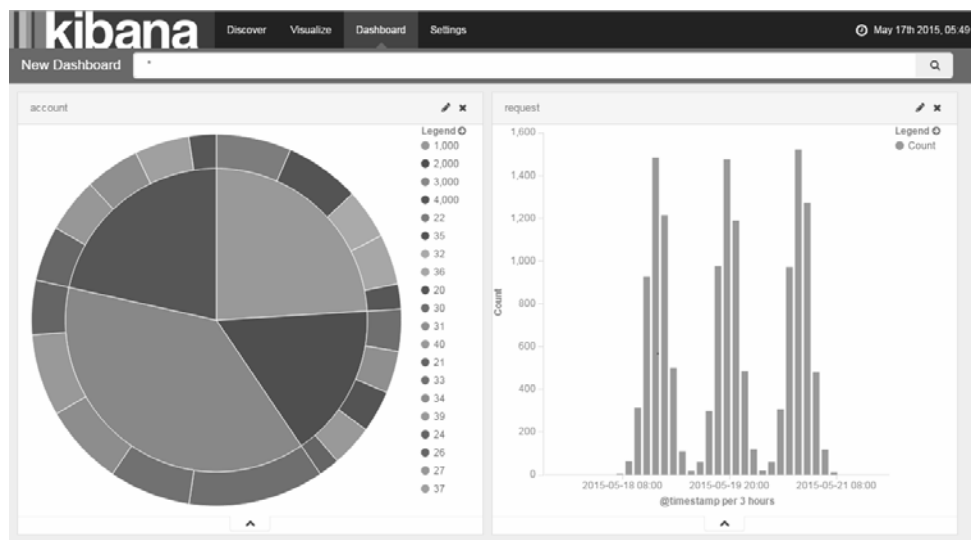


图 12-35 Kibana 主面板

我们选择一个图形的底部上拉图标，它将以各种格式显示支持图形的数据，Table 按照表格的形式展示数据，request 展示原生态的 JSON 格式，Elasticsearch 查询请求，Response 展示 JSON 格式的数据返回，Statistics 展示数据查询的一个概貌。如图 12-36 所示。

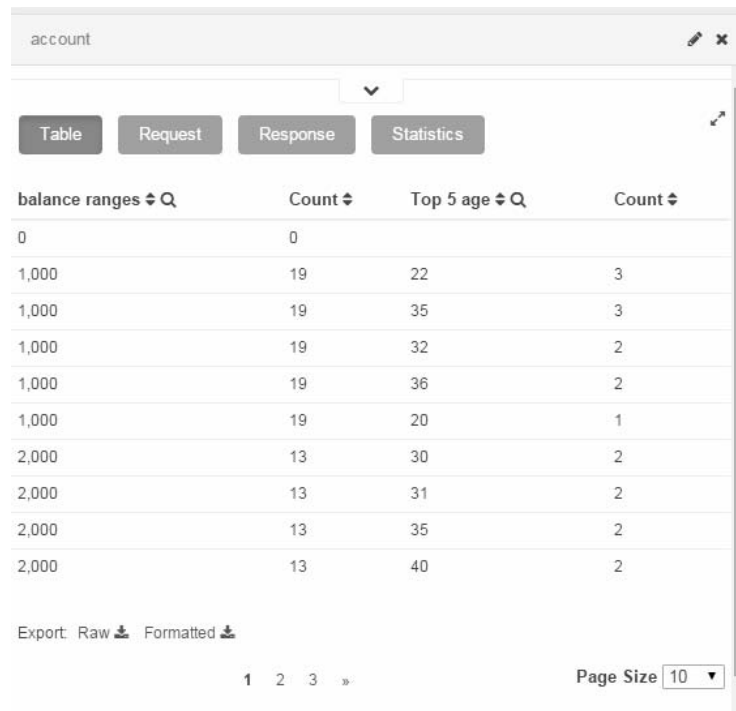


图 12-36 数据查询概貌



## 第四部分 运维管理

当运维对象达到一定规模时，复杂度会陡增，你会发现哪怕是记录简单的管理对象密码、堡垒机等信息都需要几个 Excel，何况各种复杂的配置资源。运维管理的两个重要职能是快速与稳定，既要做得快，又要不出问题，ITIL 给了我们一个参考库，在这一部分你将看到如何实现，包括配置管理库如何设计，监控如何管理，变更实施的安全性如何保证，最后我们将回到人员本身，谈一谈运维及 ESNS（企业社交系统）在运维领域的重要性。综合以上知识点，让我们思考如何将运维方法及其工具集成到 PaaS 中。

## 第13章

# 配置管理

在日常运维工作中我们有哪些常用的运维组件？它们属于什么区域？IP 地址是什么？它们的变更会影响到什么？能回答以上问题的数据，我们都称之为配置管理信息，将存储这些数据的源称为配置管理数据库，简称 CMDB（Configuration Management Database）。英国商务部出版的《ITIL 服务支持》一书这样定义 CMDB：“它是一种包含每一个配置项（Configuration Item, CI）的全部关联细节及配置项之间的重要关联细节的数据库”。

CMDB 可以说是运维管理的资源地图，它告诉我们有什么、在哪里、什么状态、如何关联。首先定义好 CI，包括物理硬件，如交换机、路由器、防火墙、服务器，也包括逻辑软件 WebLogic、Apache HTTPD、Nginx。为了准确标识 CI，需为其定义属性，如唯一性、版本、大小、状态，随后进行 CI 的组合与关联，最终形成业务视角的逻辑实体。在这些信息外面还有大量的附属信息，如登录的堡垒机、用户名密码等，建立一套完备而精准的配置管理库需要做长期的打算。

配置管理是运维管理的基石，与监控管理、容量管理、事件管理、变更管理直接关联，向后者输出有效信息，支撑数据中心运转。IDC 灾难恢复的首个系统往往不是交易系统，而是配置管理系统。在后续章节中我们还会详细说明其他模块与配置管理的关系。

在新项目、任务建设初期，为了加快整体进度，人们常常会忽略配置信息的录入，通常会在本地的一个 Excel 表格中保留相关信息，随着项目接近尾声，项目转运维进入标准化轨道后，配置信息的缺失常会引发大大小小的问题，因而必须建立一个有效的变更管理流程，在其中嵌入配置管理步骤，重视配置，保证配置先行，提前申请、登记好配置项。

在实际的运维工作中还会出现不加区分地将各类配置项录入 CMDB 中的情况。运维人员不得不做很多繁冗、重复的无效工作，手动录入低价值的数据，渐渐地，有价值的信息淹没在垃圾中。CMDB 应该只包含那些积极管理的配置项，并且通过各种方式让运维人员摆脱重复性工作，比如 CI 自动发现、提升录入界面用户体验等。

配置管理系统应保持一个持续迭代、更新的过程，有专门的开发团队支持，而不是一个固定不变的产品。小型企业可以通过 Wiki、文件夹共享等方式临时地解决问题，中大型



企业则必须拥有自己可控的配置管理系统。不应采用封闭、难客户化的系统，而应开放自由、易扩展。配置管理系统不仅用来保存数据，更注重用户体验，让运维人员可以舒服地使用，从而提升日常工作效率。

## 13.1 配置管理系统分析

### 13.1.1 服务模型进行分层

CMDB 的构建过程是一个庞大的工程，涉及的 IDC 管理团队与部门非常多，如果要将整个 IDC 所有的 CI 分析、设计进行关联并生成最终的完整的服务视图，则需要花费 2~3 年。一般很难完整地向下往上构建，比如从 IDC 的机房、机柜信息开始分析 CI，往上到硬件服务器、网络设备，最终一步步延伸到顶层的服务，如图 13-1 所示。我们可以看到 CI 处在服务模型中，我们可以对其进行分层，典型的划分方式包括：物理空间层、物理设备层、逻辑主机层、逻辑组件层、应用系统层、业务服务层，如图 13-2 所示。物理空间层包括机房、机柜、机架；物理设备层包括服务器、网络设备、防火墙、存储、加密机硬件等；逻辑主机层主要是附属在物理设备上的操作系统，比如 RedHat、Windows、ESXServer，除此之外还有与存储相关的 Nas、Lun 卷信息；逻辑组件层主要是 OS 上的一切中间件、数据库；业务系统层则是由下层组件组合而成的业务系统。业务服务层是具体的业务公司对外提供的系列服务。



图 13-1 配置管理系统功能组件



图 13-2 IDC 管理团队在分层模型中的位置

IDC 管理团队必须对 CMDB 模型分层达成共识，清楚自己所在的位置，一个管理组所运维的 CI 可能在多个层上都有体现，我们只需要找到大概的位置，比如网络小组管理的 CI，网络硬件设备属于物理设备层，网络区域属于逻辑主机层。

### 13.1.2 各IDC团队发现CI

各 IDC 管理团队在清楚自己的位置，明白上下游关系后，开始寻找自己所管理的 CI，通常有以下几种方式。

(1) 在 IDC 管理团队负责的资产清单中整理出 CI。资产并不一定由运维团队负责，有些组织中有专门的资产管理中心，从运维团队、资产管理团队日常交互的信息中可以挖掘大量的 CI，而这些 CI 一般分布在物理层面，例如服务器、网络设备、存储设备等。

(2) 在 IDC 管理团队日常工作面对的组件中整理 CI。以系统组、中间件组为例，它们每天面对得最多的就是操作系统及运行在 OS 上的各类中间件进程，运维团队整理出这些对象，并抽象出能够表示、承载这些对象的 CI。在这里强调抽象与实现之间的折中，在面向对象的设计中强调面向接口编程，抽象出所有对象的共性。对 CMDB 而言则无须完全遵循这一点，暂时忽略抽象的通用性、扩展性，而将 CI 聚焦在组织中最常用的对象上，如果你所使用的 90% 的中间件都是 WebLogic，则你就有必要为 WebLogic 建立一个单独的 CI，将它的基本属性与关联关系独立出来，同时在未来的 CMDB 界面设计上也要精耕细作，注重用户体验。

(3) 由运营应用管理人员从业务角度定义 CI。运营人员所属的业务服务条件就很好地定义了业务服务 CI，而他们直接管理的子系统则是应用系统 CI，应用系统 CI 又由大大小小的逻辑组件组成，它们之间有集群、组合关系，对这些细节暂时无须关注，重点是找到当前层面的 CI，并与下游层级做好对接。

配置管理项要与资产管理项严格区分，两种信息的用途完全不一样，配置管理项用于日常的运维，而资产管理项关乎 IT 成本核算。根据三种方式筛选出各层级、团队的 CI 后，开始对 CI 进行抽象提炼，CI 的抽象以功能、组织占用比来提炼。“硬件设备”可以看作物理设备层最高级别的抽象，我们可以将网络设备、服务器等各类具体实现映射到“硬件设备”这个 CI 之中，但这种抽象粒度的太过于宽泛会给后续的 CMDB 表示层设计用户体验提升造成很大的阻碍，我们必须以硬件功能将 CI 项进一步具体化，“PC 服务器”是运行在 X86 架构上的服务器，“交换机”负责二层网络转发，“路由器”负责三层网络数据包选路，“防火墙”负责网络安全控制，这些由功能特定并在组织内占有一定比例的 CI 单独建立。回到抽象层面，我们有必要保持一个抽象的“硬件设备”CI 来存放特殊、组织未标准化的设备，比如加密机、第三方软硬一体设备、应用安全控制硬件，这类非标设备常常因为没有合适的 CI 定义而造成信息遗漏。在 CMDB 中建立 CI 项只存在关联关系，而没有继承

关系，这与面向对象的设计截然相反，尽管我们定义了“服务器”“交换机”“硬件设备”几个 CI，但它们之间是没有父子继承关系的。在 CI 模型分层进入逻辑层后，标准粒度的 CI 应是动态的、运行中的对象，例如独立的操作系统或单一操作系统提供同一服务的单进程、进程组，而不是一个软件。不用担心对象的粒度无法组成一个完整的服务，在逻辑层面上我们可以通过定义“集群”“组合”等各种关系将这些 CI 对象拼装成合适的 CI。如图 13-3 所示。

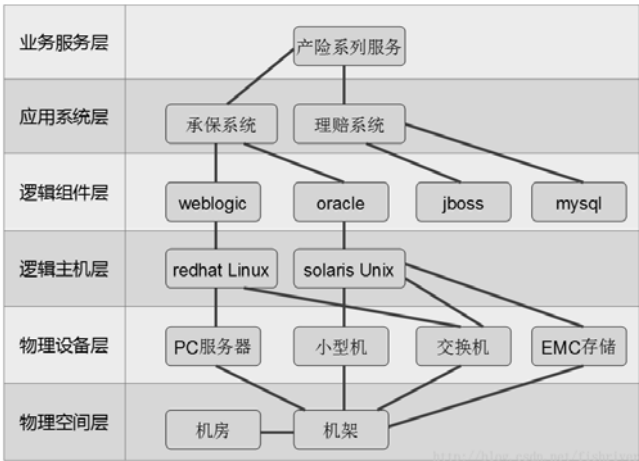


图 13-3 各管理团队寻找自己的 CI 项

13.1.3 IDC 管理团队定义 CI 属性

很快我们手上有了一批合适的 CI 清单，我们还有大量的数据来说明 CI 是什么，这些数据可以用来作为 CI 的属性，我们有什么方法从这些数据中发现属性信息？我们将 CI 的属性分为 6 类（如图 13-4 所示）：核心、附属、控制、服务、关联、扩展。下面我们对这些属性类别逐一进行说明。

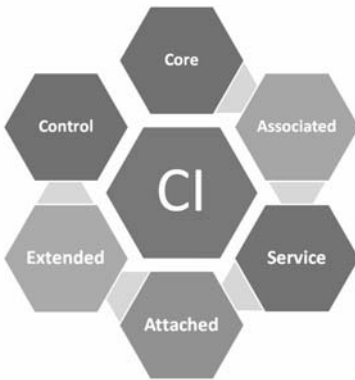


图 13-4 CI 的属性

### 1. 核心 (Core)

在运维工作中操作 CI 时经常使用的信息具有唯一 ID、命名、生命周期、易变化等特征。核心属性维持了 CI 在整个运维管理工作中的有效运转，其中的每个属性都是必要的。

- 唯一 ID：表示当前 CI 在 CI 对象集合中的唯一性，数据库表字段 `unique id` 递增序列保留增长容量空间，可以符合大部分 CI 属性的要求，唯一 ID 与 CI 本身所代表的实体没有关系。
- 命名：命名与唯一编码的不同之处在于命名是在前台可见的，它是印在 CI 实体内容之中的，命名本身就包含了唯一标识，同时附带其他说明。我们所说的印在主要指逻辑层面，比如一个操作系统命名为 `CNSZ031415`，一个 `WebLogic java` 进程加入 `SERVER` 环境变量 `chs-coreStg7891`，这个名称是“活生生”的，可以通过程序在运行态的实体中动态寻找，这为以后的自动发现、自动化运维等工作埋下了伏笔。命名内容是组装而成的，我们应当精简命名，命名是我们最关注的，其他关注点在单独的属性中体现，不同的层级的关注点不同。“`CNSZ031415`”“`CN`”表示中国，“`SZ`”表示深圳，“`03`”表示测试环境，“`1415`”是主机层面上的序号分配，以上命名信息解码可能是一个系统管理员所关注的。而中间件层面与应用系统联系得比较紧密，因此在命名中要体现所属的应用系统，在“`chs-coreStg7891`”中，“`chs-core`”是子系统名称，“`Stg`”是环境名称，“`7891`”则是序号。

生命周期：一个 CI 项也有着从摇篮到坟墓、从自然中来到自然中去的过程，不同时期的管理级别有着不同的要求。生命周期不仅仅作为一个状态而存在，还作为一个期限而存在，在资源管理中为了保证所有资源合理而有效地利用，常常定义一个资源“释放”期限帮助我们对 CI 资源进行管理。

### 2. 控制 (Control)

控制属性与 CI 本身也没有关系，它关注权限、审计、版本等控制方面，这些属性在整个 CMDB 中是通用的。值得我们关注的是权限、审计、版本这三个控制属性映射到数据存储对象后的位置截然不同。

CI 的操作权限是受用户所属角色、组织、单位限制的，我们在 CI 元数据中定义此属性，但此属性不是说明我属于谁、属于那个组织，而是说明我是谁。属性表示的是“我是银行的服务器”，通过后面的关联，我们可以导航到“我是一个跑着 `WebLogic Java` 进程的服务器”“我是一个为银行信贷应用服务的 `WebLogic Java` 进程服务器”，在这些属性说明中没有做那些不该做的事，仅仅说明我自己。权限的切入点不能放在 CI 之中，比如在 CI 中定义“银行”“信用卡服务组”“李四”这样的组织、人员信息以标识它的权限，这样的设计会让后续 CMDB 的维护难度大大增加。权限的切入点要放在“行为”“属性”两个层面，对于服务器的读写权限中的读写是一种“行为”，我们应该做一个行为的拦截器对权限进行控制，这是放在 CI 之外的，而“属性”层面则是通过属性内容由“规则”去自行判断的。规

则控制这权种限，“规则”有时剥离在 CI 之外的。“我是一个为银行信贷应用服务的 WebLogic Java 进程服务器”在这个 CI 配置项的表述过程中表达的是自己的属性，而“只有为银行应用服务组服务的基础架构人员才能修改银行机器，且只能修改 WebLogic Java 进程的 PC server”是一条规则。规则是权限控制的逻辑，而权限控制的输入是当前操作人员的所属属性、CI 控制属性，输出是“你是否可以操作”。CI 的控制属性不完成权限控制工作，它是权限控制实现的输入元素之一。

审计告诉我们谁在什么时候新增、修改、删除、甚至查询了 CI 对象，CI 的审计信息量视情况可大也可小，审计功能可以做成定制化开关，控制数据库容量。为了做好版本控制，如果每次对 CI 的变动都保存完整的 CI 信息及其所关联的 CI 当前版本信息，则这会大大增加开发复杂度及 DB 数据存储量。从经验上看没有必要在所有的 CI 上定义版本号，而是通过“变更快照”“审计对比”功能中附带的完整版本控制工作。变更快照帮助我们对操作 CI 的上下游配置信息进行快照，并保存在当前的变更流程中，由于 CI 变化只与变更有关系，因此在变更流程中启动快照，在数据库结构化信息之外保存 CI 信息，即可完成对未来异常问题的查询。另一种做法是在开启了审计功能的 CI 上记录每一步修改的内容，根据这些修改信息我们同样可以完成对故障信息的收集及版本回滚。

### 3. 服务 (Service)

运维管理工作最终提供给用户的是服务，而不是一台主机、一个进程。服务属性与 CI 属性也没有直接关系，而是与服务流程相结合，分析关联影响。你可能要问一个属性怎么来识别分析影响？首先，服务属性一般在分层模型中的应用系统层，例如我们对“承保系统”定义：一级核心、运维时间为 23:00~24:00、是否冻结等，这些属性都是围绕着服务、应用而言的。下一步我们会分析 CI 之间的关系，这些服务属性是向下继承的，当我们变更操作一台服务器时，进入变更流程后就会自动发起关联影响分析，你会收到询问信息，例如“你确定要维护这台机器吗？它是承保系统，已经冻结变更，并且维护时间只有一小时哦？”真正的变更影响分析是通过服务属性、CI 关联、属性继承来进行的。当然，你可以完全忽略这些 CI 关联影响评估信息，但金融类 IT 运维以风险控制为重点，变更安全粒度的要求非常严格，什么时间点可以做什么事要求通过 CMDB 中的数据进行分析，输出一份准确报告。

### 4. 附属 (Attached)

这些属性一般是可选的、固定的，与实际运维工作无直接联系，仅仅是帮助我们更进一步地了解 CI。例如 CNSZ031415 服务器的 CPU 主频是 3300MHz，这个主频就是附属属性，附属属性由于其稳定性，常常批量导入、默认设置。

### 5. 关联 (Associated)

CI 属性中一个非常重要的种类是关联，它表示 CI 之间以一种什么样的关系组合在一

起，形成一个面向服务、应用的整体。我们认为 CI 之间的关联也是一种属性，这类属性具有双向导航的特性，可以由一个 CI 找到上下游、平行层级的关联 CI。

### 6. 扩展 (Extend)

扩展属性是为一线运维人员服务的，它们以文档、图形、模板、服务目录、应用档案等非结构化数据方式组织起来，关联到外部的知识库、运维规约、应用档案，告诉一线运维人员操作方法、生产风险和一般规则。

## 13.1.4 确定CI之间的关联

确定 CI 之间的关联关系的过程也相当具有挑战性，关系的设定决定了以后在使用过程中对配置项的快速定位与影响度分析的有效性。在不同的 CI 之间会有不同的关系类型，在相同的两个 CI 对象之间还可能有不同的关系，而且关系的确定也需要有适当的粒度，关系的保留采用“有用性”原则，即对业务来说是有用的。忽略那些对业务不产生影响的关系类型。CI 之间的关联关系类别限定在四种范围内：组合、集群、使用与依赖，除此之外再无其他关联关系。

CI 之间通过关联关系最后组成一件完整的“产品”，我们不会像对大多数 ITIL 软件一样对这个产品分解成层次结构型体，提前定义一个 CI 所属层级会让 CMDB 失去灵活性，在很多情况下一个 CI 在不同的产品中属于不同的层级，这里的层级与服务分层模型层级是不一样的，这里的层级是相对于最终拼装成的完整产品而言的。产品的结构层级对于工程领域产品配置项来说非常有效，一个产品的组成部分是固定的，而产品组成的元素关系相对于虚拟软件而言更接近现实世界，大部分可以采用依赖关系。而 IT “产品”很难用工业产品标准化的方式将服务、应用、进程、OS、硬件资源组合起来，不同的应用有着不同的组成元素，并且随着 IT 的演变，组成组件的类别与关系也在不断变化。另外，即便是同样的组成形式，IT “产品”组成元素之间的多重性也在变化，在应用访问量预计大增时，应用服务器集群的数量会陡增，我们没有必要以产品层级的方式将 CI 固定起来，而应以一种灵活而零散的方式自由组合 CI。

### 1. 依赖

表示一个 CI 的存在强依附于另外一个 CI，依赖追求 CI 之间完整性，禁止 CI 独立存在后产生的垃圾数据。在进程与 OS 之间就有这样一种依赖关系，在添加一个 WebLogic 进程实例时我们要求必须录入它所依赖的 OS，进程脱离了所运行的 OS 则没有任何意义。依赖关系也会产生一些问题，由于强依附关系的存在，CI 链可能因为某一个 CI 的缺失而导致整体信息的录入延时。回到这个例子，WebLogic 和 OS 对象在不同管理团队中运维，中间件组录入 WebLogic 信息时强依附到系统组的 OS，而系统组不一定录入了 OS 信息，因而中间件组也无法录入。必须在变更管理中建议有效流程，保证相互依赖的 CI 信息的有序录入。

## 2. 组合

表示一个 CI 由哪些其他 CI 组成，是一种“整体与部分”的关系。组合关系相对于依赖的强制性要稍微弱一些，可以独立创建一个 CI，之后再陆续补充它的组成 CI。一旦组合关系建立，下层被组合 CI 的生命周期就与上层 CI 联系上了，在这里产生了一种强制关系。车险承保系统由 WebLogic 中间件组成。我们可以单独创建车险承保系统，后续再添加 WebLogic 进程，一旦这种关系建立好了，它们的生命周期就统一起来了，如果要下线车险承保系统，则必须先下线下层的 WebLogic 进程，这种生命周期的传播是从上至下的。组合关系适用于下层 CI 唯一服务于上层 CI 的情况。

## 3. 使用

标识一个 CI，将另一个 CI 作为资源 CI 使用，这种关系是松耦合的，Oracle 数据库使用了一个存储 Lun 卷，即便 Lun 卷不存在，Oracle 数据库还是能够运行，它可以将数据存储到其他卷上。在现实世界里不会出现 CI 之间完全松耦合的情况，提出这种关系是为了保持 CMDB 的灵活性，在无法区分组合、依赖关系时，我们会将 CI 之间定义为使用关系。

## 4. 集群

集群是 IT 领域常出现的概念，它表示一组对象共同完成一个任务，对象之间有主备、并行、优先级等进一步的关系，集群后形成一个逻辑 CI，比如 10 个 WebLogic 进程集群对外提供服务，它们是并行的，两台 ESX vmware 设备互为主备等。运维领域的集群会引入对负载均衡、容错、高可用等术语的讨论。集群是 4 种关系中比较特殊的 1 种，伴随着关系的建立，集群（Cluster）中的每一个成员（Member）之间还有亚层次的关系，如果是优先级，则成员都有一个级别属性；如果是主备关系，则成员会标明为主或备。这在设计 CMDB 数据模型时要提前考虑。如图 13-5 所示。

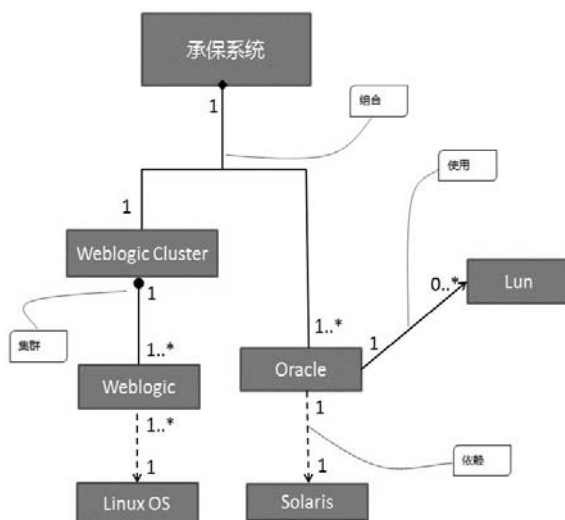


图 13-5 CI 关联关系的例子

读者可能会好奇，为什么关系只有这 4 种？下表列出了 CMDB 中一些常用的关系，现实世界中的关系可以被描述成很多种。CI 与面向对象设计不一样，CMDB 并不强调父子继承关系，即便有这样的一层关系，也会通过定义两种不同的 CI 实现。而对于其他各种关系的描述我们可以映射到“依赖”、“组合”、“使用”与“集群”上，如表 13-1 所示。

表 13-1 CI 关联关系列表

编 号	关 系	说 明	示 例	备 注
1	安装在..上	Install on	数据库安装在主机上	依赖
2	连接关系	Connect with	主机与网络相连	使用
3	依赖关系	depend on	应用依赖于中间件	依赖
4	父 CI	parent	进程与它的一个具体实现，父是进程（Process）	不使用这类关系
5	子 CI	child	进程与它的一个具体实现 WebLogic，子是 WebLogic	不使用这类关系
6	物理关联	associate with	主机与存储关联	使用
7	文档关联	with doc	某软件有某文档	使用
8	使用关系	use	谁使用某 PC	使用
9	监控管理	admin & monitor	谁监控某机器	使用
10	组成关系	consist of	销售系统由主机、DB 和中间件组成	组合
11	运行于...上	perform on	应用运行于 OS 上	依赖
12	备机	standby	主机 A 是 B 的备机	集群

在建立好合适的关系后下一步是确定关联对象之间的多重性，它用来指出可被允许生成的关联 CI 的数量，即最多可以生成多少数量（上限）、最少不得低于多少数量（下限）。关联的两端以“下限..上限”的格式标示出多重性。星号（\*）代表不指定上限，下限最低为 0。CI 的关联关系、多重性确定后，也就完成了 CI 之间的导航功能，我们可以根据一个服务 CI 生成该 CI 所引用的资源树。

除了以上 4 种关系，还可能有一种独立的系统层面的关系，比如 A 系统调用 B 系统，这种关系有助于我们生成应用之间的服务架构图，这种关系不在配置管理层面，如果要做成一套大而全的系统，对这种关系则需要另外考虑。

## 13.2 配置管理系统设计

CMDB 已拥有一个非常成熟的市场，可以采用商用、开源、自主研发三种方式来实现我们的 CMDB 系统。从如图 13-6 所示的结构图来讨论配置管理系统设计的三个方面：用户界面、数据模型、附属功能。



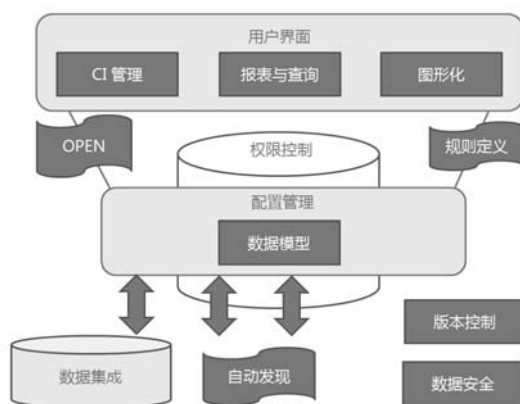


图 13-6 配置管理系统设计组件

### 13.2.1 用户界面设计

用户界面直接面向配置管理员及其他 IT 运维管理人员，他们有职责定义 CI、管理与使用 CI 属性并确定他们之间的关系，并与运维操作、管理流程相关联。配置管理系统不是用来存储冷冰冰的数据的，在运维管理中，配置管理、监控管理系统的使用频度要远远高于其他系统，在界面上关注用户体验是无形中提升运维效率的方法，好的 IT 运维经理应该时刻关注一线人员的情况。

#### 1. CI 管理

配置管理员、IT 运维管理人员对 CI 的操作，可以简单地认为只是日常简单地增、删、改、查。用户体验的提升就潜藏在这些基本功能之中。

对于复杂的一屏下来几十个字段的 CI，我们应该分屏、向导式地一步步引导运维人员录入数据。具体说来，在进行 CMDB 界面设计时，要通过各种视觉手法来告诉用户：这个向导过程总共有哪些步骤、用户已经完成了多少步，以及还有多少步未完成等。这些要求其实是任何信息导航设计都要达到的一些基本目标。如果一个界面设计达到了这些目标，那么它就能使得用户具有一种对整体的把握感。用户在看到自己所完成的每一步所带来的进展，以及在整体上一步步地向成功接近时，他就会更加有信心并愉快地完成整个任务。在向导结束前，将用户的输入以某种方式显示给用户以便确认，并且越及时越好。这一准则应用了反馈原理。该原理指出，任何同人进行交互的系统，都需要将其内部状态以某种方式表现出来，尤其是要对用户的输入动作进行反馈，只有这样，用户才能知道系统是否检测到并接受了自己的输入，以及自己的输入是否正确，这样用户才能进行相应的后续操作，比如调整自己的输入或改正错误。缺乏反馈的系统会让用户感到茫然和焦虑。向导式界面是一种着眼于帮助人们处理复杂任务或不常见任务的交互策略。为了更有效地达到上述目标，我们需要对其中的很多交互细节加以认真处理。对于常用的输入，应尽可能采用

自动补全的方式，运维管理人员输入部分有效的信息后就能自动查找、关联到所需 CI。CI 属性的选择是有规则限制的，比如一个 WII 区安全区域的 WebLogic 就无法运转在 APP 区域的 Host 上，这些规则要提前定义好，并提早过滤掉干扰选项。在网页上减少全屏刷新，通过 Ajax 局部范围的更新内容，递进式地按层级显示数据。各 CI 管理菜单应按分析出的服务分层模型归类。无论底层的数据库 Schmea 如何灵活的设计，也没有办法将 CMDB 前端完全产品化，用户体验的提升是在配置管理系统反复迭代的开发过程中实现的，IT 运维团队应当有一个专职的开发工具团队在后端支撑运维工作。如图 13-7 所示。

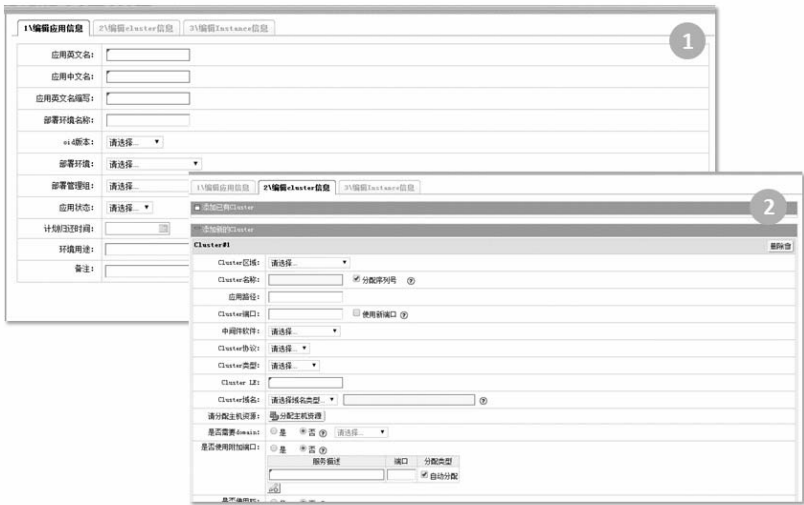


图 13-7 资源配置界面

2. 报表与查询

报表功能可以实现依据不同的属性组合进程查询、检索 CI 集合，定制 IT 运维管理所需要的报表清单。在 CI 查询上，由于 CI 的类别特别多，所以要提供一个类似于搜索网站统一查询的页面，按大类划分，无论是哪种 CI 关键字，都可以在一个查询界面中搜索到结果。如图 13-8 所示。



图 13-8 CMDB 搜索界面

3. 图形化

配置管理库的一个基本功能是将 CI 之间的关联关系图形化地展示出来，可视化应当对

CI 对象进行树形结构展示。另外，CI 集合还要能最终组合成 IT 系业务视图。CI 可视化的另一个重要功能是支持网络拓扑，依据 IDC 情况对网络拓扑分层展示。下面是一个三层网络拓扑展示功能。

#### 1) 三层展示核心网络骨架

第一层：数据中心（IDC）、职场、租用机房。

第二层：安全区域、核心区域、其他。

第三层：防火墙、核心交换机、F5、其他设备。

层级关系：上层实体可以分解成下一层。例如第一层中的 IDC 实体有第二层中的安全区域、CORE 组成。

安全区域的定义：在同一防火墙后被保护的区域，使用单一的功能区域（通过路由层 ACL 保护的区域也属于安全区域，定义在一个子集内）。

关联关系的定义：定义出所有图层实体之间的关联。比如直连、专线、cn2 等。由于要生成图形，所以会有关系是否有方向的定义。

最后一层的拓扑图：进入第三层后是详细的拓扑图。

#### 2) 图形与配置管理数据相结合

图形中的所有实体、关联都可在配置管理系统中查看。我们的所有核心实体都从 CMDB 中来，例如防火墙、汇聚层交换机、F5 等。

在配置管理中要体现实体之间的关联及关联的方向。

在配置管理中要体现实体之间的包含。

图形是通过配置管理数据中的属性自动生成的。

#### 3) 图形的检索和定位

由于与配置管理相关联，图形是可以检索和定位的。

主要检索的字段有网络设备名、保护网段等。

通过检索网段能够查到安全区域，并可以查看拓扑图，这样便于确认终端之间的逻辑路径及经过的防火墙。

## 13.2.2 权限控制、规则定义和OPENAPI

### 1. 权限控制

权限是服务于整个运维管理系统的，要考虑 CMDB 中的数据模型是否适用于当前整体

的运维管理系统。

权限系统分为授权对象、保护对象。授权对象一般指被授权的用户，如果所有的用户对保护对象都关联一条权限，则会产生大量的冗余数据，这对于系统管理来说是一件痛苦的事情。为了解决这个问题，我们在用户的基础上创建角色，将一组权限集合归为一个角色并赋予用户。

保护对象有功能、数据两类，功能对象包括菜单、按钮、操作，这类对象可以是对一般的 CI 对象操作的保护，也可以集成到运维流程管理控制中。数据包括 CI 类型、条目，要记住，我们保护的不仅仅是动作，还包括数据本身。要识别数据的保护级别，可以通过 CI 类型与单个属性进行设置。如图 13-9 所示。

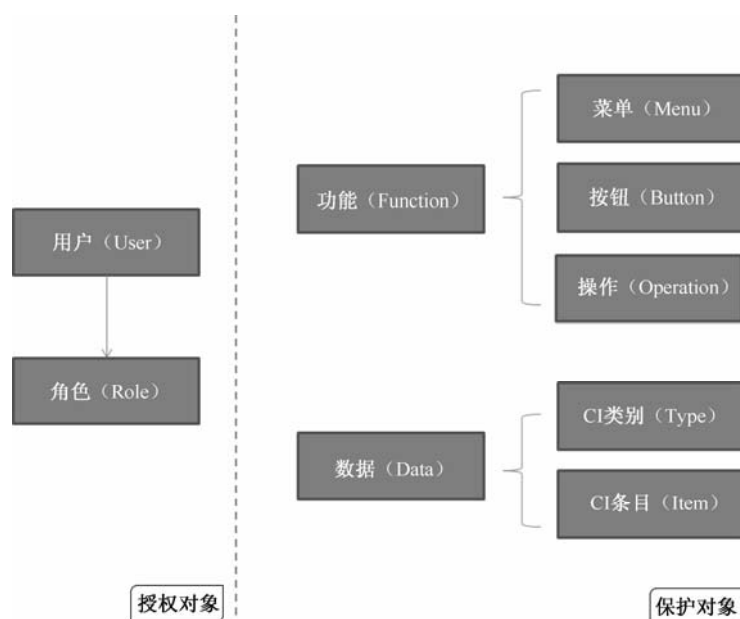


图 13-9 权限控制功能组件

## 2. 规则定义

在 CMDB 中也需要规则，这个规则用在对 CI 对象输入的限定上。“如果是 WII 区域的 WebLogic，那么它所依赖的 OS 也必须在 WII 区域”，“当 OS 的状态为已上线时，必须输入该 OS 的应用管理员”“主机名是按 ‘cnsz’ 加上数字的规则定义的，输入的主机名称不符合规范”这些都是基于属性的规则，我们在输入一个 CI 对象的所有数据时，要先通过规则来验证该 CI 输入是否正确。如果 CMDB 具备一个有效的动态的灵活的规则引擎，则 CMDB 数据录入的准确性可提升一个级别。

## 3. OPENAPI

OPENAPI 有利于与内外部系统的 CI 数据的集成汇总，企业通常会采购一批完整的 IT

组件设备，例如 VMware 的虚拟机管理，我们可以通过 OPENAPI 将数据由 VMware 导入到 CMDB。除了完整组件同步，对于自动发现的 CI 导入一样提供了标准接口。与直接通过数据库层面导入数据相比，OPENAPI 在规则、权限之前对数据的精准性进行了更严格的控制。

### 13.2.3 数据模型的设计

前面是我们在企业、组织内对 CI 分析的整个过程，并且对用户 UI、权限、规则及对外开放等都提出了设计要求，那么最核心的 CMDB CI Schema 应当如何设计？我们将问题分解为关键的 CI 数据模型、权限模型，首先我们将针对 CI 数据模型的设计进行讨论。

CI 数据模型设计有两个极端。

将所有的 CI 放入一个 CI 表中，我们称之为巨婴表。这个表要解决的第一个问题是不同类型的 CI 有不同的属性，随着 CI 类型越来越多，这个表的字段要无限扩张，不同的数据库对表字段数量是有限制的。第二个问题则是将所有的 CI 放在一个表中，其性能会出现瓶颈，特别是要求保证事务一致性时。第一个问题的限制是硬性的，而对第二个问题我们可以通过多种手段缓解甚至解决，例如使用缓存提前预热数据，建立灵活的外部索引表，对于报表数据提前一天夜间运行等。第一个问题的解决方式是在巨婴表上预留字段数，或者在每次加入新属性时加入新字段，问题逐渐演变成我们到底是该使用面向行的关系型数据库，还是该使用底层用户分布式的面向列的数据模型。在这里笔者尝试过使用 HBase 存储 CI，并采用面向列的解决方案，最终得出的结论是我们的 CMDB 不适应这种存储方式，我们面向的列是伪列，是把所有 CI 拼装起来的列，其本质仍旧是面向行的。

好了，让我们从一个极端走向另一个极端，为每一个 CI 建立一个表，在不断变化的 CI 关系中创建关联表，服务器是一张表，OS 是一张表，交换机又是另外一张表。这可能走向了另一个极端，可称之为芝麻表，整个 Schema 设计下来动辄几百张表，让日后的开发人员根本无从维护。或者说设计这批 Schema 的维护人员也会陷入设计的痛苦之中，这里最怕的是寻找边缘数据，对于那些在日常工作不经常使用的对象，我们会问：“它在哪里，它是什么？”

分析强调的是对问题的需求和调查研究，而不是解决方案。我们在前面做了大量的分析工作：分层模型、CI 查找、属性定义、关联 CI，我们需要依据在分析工作中整理出来的信息完成设计。设计强调的则是满足需求的概念上的解决方案。在两个极端设计方案之间我们选择了平衡。

在巨婴表的基础上，我们拆分 CI 与属性、CI 类型与属性类型，把它们分解为 4 张表。对于芝麻表，我们用 CI 类型控制芝麻表的产生，不允许 CI 的某一具体定义一个独立表，ER 实体最终如图 13-10 所示。

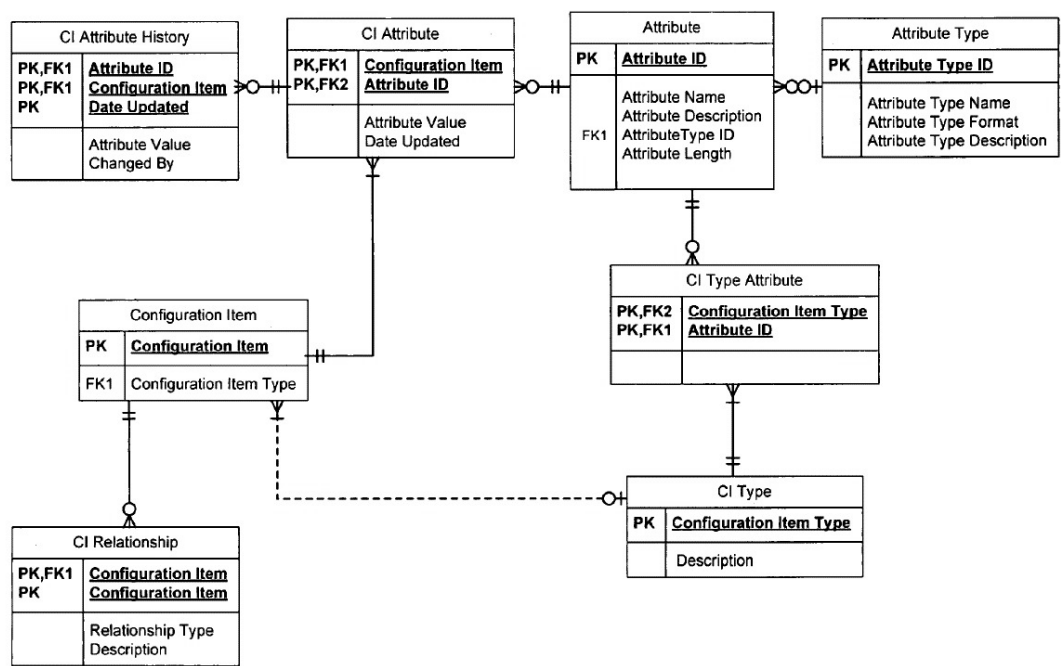


图 13-10 CI 的 ER 实体图

我们用 8 张表保存 CI、CI 属性、CI 关系及版本信息。相对于芝麻表，我们将核心对象、边缘对象统一放入这 8 张表中。对于特殊的 CI 资源，你会问是否也要套用这个基础表模型？笔者的答案是如果特殊 CI 资源不属于边缘数据，而是在整个 IT 组织中都需要使用的，时刻被关注的，例如 IP 资源，那么我们为什么不在这简单的数据模型上做进一步的扩张呢？

CI\_Type、Attribute、Attribute\_Type、CI\_Type\_Attribute 4 张表的作用放在面向对象编程中可以认为是类的存储地，例如一个人，它定义了“人”这个 CI，以及“人”的属性，有名字、性别、身高。Configuration\_Item、CI\_Attribute 则是具体的对象，还是以前面的“人”为例，在这两张表中存放的是“小明”这个人，姓名是小明、性别男。CI\_Relationship 存放的是 CI 间的关联关系，CI\_Attribute\_History 存放属性的版本信息。

在讨论完 CI 数据模型的折中设计后，再回过头来看看权限模型。在权限控制那一节我们已经谈过权限设计模块基本元素。权限的授权对象是固定的，一般有用户和角色。用户、角色之间是多对多关系。而保护对象是不固定的，一个按钮、菜单、数据项等都是保护对象，而保护行为一般可固定在是否可读写上。谁对什么可以有怎样的动作，这就是一个权限的判断规则，who、what、how 三个元素组成的一条规则。who 固定在用户、角色之上，what 不固定，how 除读写外也有其他动作类型。

最终的权限控制放在了一张表中，表的字段如图 13-11 所示。

PrivilegeID
Privilege_Auth_Type
Privilege_Auth_Value
Privilege_Protect_Type
Privilege_Protect_Value
Privilege_Operate

图 13-11 权限控制表字段

Privilege ID 是主键，Privilege\_Auth\_Type, Privilege\_Auth\_Value 关联到用户或角色表，例如这条权限规则是用户的，则 Privilege\_Auth\_Type 指向 user，而 Privilege\_Auth\_Value 是 user\_id。这两个字段代表了规则的 who。Privilege\_Protect\_Type、Privilege\_Protect\_Value 则代表了保护的對象，如果我们保护的是数据 CI，那么 Privilege\_Protect\_Type 标识为 CI，Privilege\_Protect\_Value 标识为 CI 的 ID，注意，我们在数据保护上可以对一条 CI 进行保护，也可以对一个 CI Type 进行保护，最后我们看到的是具体保护的行为是什么，在这里用 Privilege\_Operate 表示。

## 13.3 配置管理数据准确性的保证

### 13.3.1 识别CI的OWNER

要想保证 CI 的准确性，首先得找到谁使用 CI、谁维护 CI、谁对 CI 负责、CI 的准确性与谁的 KPI 挂钩，而不是在大家需要使用 CMDB 数据时才抱怨。在 CI 的使用者与 CI 的维护者对数据的准确性要求不一致时就会出现抱怨。这两个角色可能出现在一个团队中，也可能在不同的团队中，系统服务组的资产负责人是 CI 的使用者，而系统服务组的运维负责人是 CI 的维护者，他们在同一个团队中，他们也会因为数据的不一致而产生矛盾。跨团队关于数据不一致的问题频率可能更高，CMDB 的某条数据的准确性可能是无关生死的问题，但如果 CMDB 的大部分数据都荒废而陈旧，则后面的 CMDB 管理工作则如大厦将倾、岌岌可危，没有人再对 CMDB 负责，最终为了数据的准确性而定期做一次“资产盘点”，在重要时刻数据用不上，每隔一段时间还要增加工作负担，事实上推动另一个团队的人员及时录入一条“无关紧要”的信息非常难。

我们第一个要识别的是 CI 的 OWNER，CI 的 OWNER 对数据的准确性负责。他们不一定是数据的维护者，他们更多的是承担起对数据准确性的审计，受理数据异常问题，推动数据维护者及时更新和修正数据。对于数据维护者而言，他们的 KPI 与他们维护的数据准确性是直接挂钩的，数据维护者的维护动作一般是嵌入在某个变更流程中的。被升级出来的数据异常应以变更质量问题处理。

### 13.3.2 识别CI的生命周期、关联运维流程

将 CI 的维护关联到运维流程中是保证 CI 数据准确性的另一手段。要将 CI 的维护关联到运维流程之中就必须先识别 CI 的生命周期。

不是所有的 CI 都需要建立生命周期，生命周期的出发点是保证数据准确性及对资源的有效利用，只有那些资源宝贵、功能独一、具有明显生存期的 CI 才需要建立。比如硬件服务器资源、SAN 存储卷资源是很宝贵的。一条生产专线的建设本来就分为提交、勘察、建设、测试、上线等生命周期步骤。对于这些特殊的 CI 我们应当识别出来，定义其生命周期属性。

下面是一个典型的硬件系统生命周期。

- 申请：每个生命周期都是有起点的，它可能是由一个运维申请流程引发的，这时我们定义了一条空数据的 CI 对象。
- 受理：申请必须被授权以确保它符合预算和其他要求。大多数组织都有正式的程序对于新技术要求授权。
- 安装：若申请的设备符合规格，那么运维服务支持团队开始对它进行安装部署工作，这部分内容是嵌套在运维流程中的。
- 测试：对新安装部署的设备进行测试，确认它满足生产运行的要求。
- 上线：一旦设备经过了前面的几个步骤，则正式上线运行。
- 下线：当设备不再被需要或无法满足我们的要求时，安排其下线。

我们在不断地询问，下一步做什么？在某些情况下我们可以设置一个定时器，或者依据一些依赖条件进行自动检测，不管是定时器还是检测，它们所要完成的任务都是“提醒”我们要进入生命周期的下一个阶段了。定时器适用于租期到期的情况，而依赖条件则适用于检测上层依赖资源是否存在，比如若一台已上线的服务器上没有应用了，则 CMDB 的报表功能会“提醒”我们是否下线该服务器。

CI 的生命周期可以嵌入运维管理流程中，在一个具体的变更步骤中可以定义是否修改一个 CI 的哪些属性。

### 13.3.3 数据有效性的审计

在确认 CI 的 OWNER，并将 CI 生命周期嵌入运维流程中后，可能还是无法保证数据的正确性，因此还需要建立起配置管理审计机制来保证数据的准确性。

审计工作发生在日常的运维中，CMDB 数据的使用者发现数据异常后需要有途径立即



受理，记录下问题后更新 CMDB，而不是置之不理。配置管理审计类似于资产的盘点，企业应根据需要设置周期，一般一年至少开展一次。另外，CMDB 还可以就一定的范围进行专项审计，从而小范围、细致地核查某类 CI 或某项关键服务所涉及的 CI “账实相符” 状况。CMDB 审计发现数据不符时应尽快查明原因，并通过变更工单提请变更，最终修改 CMDB 数据。CMDB 审计流程应该独立展开，审计员应由专门成立的监管部门担任。

### 14.1 运维监控管理的问题与价值

你是一名企业应用 IT 管理员，在一个阳光明媚的早晨，你兴冲冲地来到公司上班，你所运维的应用系统面向 3000 名用户，突然间系统的某一功能不可用了或性能变得非常差，无法满足用户的需求，用户的投诉电话连连不断，上级领导要求尽快恢复并查明原因。你问自己这些异常为什么一点征兆都没有。你确定昨天晚上什么也没做过，而且你很自信对应用系统的构成非常熟悉，你尝试在脑海里回顾可能的问题点，你召集了应用系统构成组件的运维管理人员，大家登录各自的管理设备上执行命令，并且每个人都回复“没有问题，一切正常”。一轮轮分析诊断下来终于找到了原因，挂载在主机上的某一个存储卷的访问响应时间突然变得非常慢。你看了看手表，大半天就这么过去了，你再看看会议室中聚集的人群，还在疲惫的分析中。会议上领导狠狠地批评大家，为什么这么久才定位到问题？为什么没有做好应该做的监控？为什么没有相关数据趋势的对比？这背后确实因为异常而造成了实际的业务损失。在一家全国领先的大型金融 IT 机构中，一个小小的失误、几分钟的中断都可能造成难以想象的经济损失。

我们对外部客户提供的是服务，服务的运行由各应用系统来支撑，而一个应用系统则由底层基础资源构建，它们可能是一组运行在 Linux 服务器上的 WebLogic 进程，它们的权限认证由一个 ldap 服务器负责，系统域名则由一组 DNS 服务器来解析，它可能还会挂载 NAS 存储卷，数据存储在后端的 Oracle 中，这些组件的通信还要经过中间的网络设备，可能会有交换机、路由器及防火墙，这里我们仅仅考虑一个独立运行的应用。我们看到一个应用的健康运行与 IDC 的很多组件都有关系，而这些组件的健康指标又有哪些呢？目前服务器的 CPU 负载是多少，历史高峰期又是多少，磁盘使用率是多少，网卡流量是否正常，内存使用率是多少，SWAP 区域是否够用，打开文件数有没有超过限制，使用的网络端口数是否达到了极限，vanish 缓存命中率是否正常，安全管理进程是否全部启动中，MySQL replication 是否同步了，数据是否一致？这里仅仅是看一个 OS 及其上的一些对象，一个 OS 是节点，一个大型 IT 公司 IDC 的监控节点数量在 10 万以上，而作为健康指标的监控项数

量会很轻松地突破千万。

### 14.1.1 监控管理的无形价值

运维岗位的价值体现在无形，不能出故障，不能出问题，在整个 IT 生态圈中要看不到运维的存在，这就是运维管理的最高境界，而“存在即合理”，看似不存在的也会看似不合理，即看似没有价值。优秀的运维人员处在一个极其复杂的矛盾中，但优秀的人员往往对事务的美好是有向往与追求的，他们会想尽一切办法驯服业务层面下的形形色色的基础资源，如同压制妖魔鬼怪一样不断提升运维技能，他们仍旧会螺旋式地追求无形境界。在追求的过程中实现运维价值。

怎样才可以保证服务质量？你可能会说高可用（High Available），在十年运维生涯中，我看过最复杂的高可用应用“架构”，应用服务从上至下都是双备、双活的，前端负载均衡是主备，应用层采用 Ejb Cluster 集群，主机网卡都是双网卡绑定，主备虚拟机存活在不同的物理宿主机上，编制了特定的规则来严禁主备机器跑在一台服务器宿主上。我们假设的前提是应用本身是无法承受任何错误的。在追求完美的高可用时，弊端自然而然生，对于不同级别的应用底层资源的组成结构有很大的差别，为了识别、定制、客户化这些零散差异，底层资源的交付速度大大降低。在完美高可用的架构下运维管理的复杂度也会相应提升，你可以说提前定义好所有的规则通过自动化来实现，应用对底层资源的要求差异化太大，标准化、自动化定制很难形成。高可用带来的另一个问题是运维成本的提升，在长期的运维经验中，我们终于明白了应用可用率的 99.99%，为什么小数点后面的 9 位数越来越长，实现的成本陡增。

保证服务质量与稳定性的另一个利器就是这里主要讲的主题了，即一套完善的监控平台。运维是无形的 IT 价值，则监控是无形的运维价值。监控职能如同运维一样处在一个比较尴尬的位置，告警节点与告警项的维护本身就是一大难题，告警节点是否有遗漏，告警项是否合适，我们是否有足够灵活而快捷的监控平台来帮助我们应对各种各样的监控节点，以及是否有一个快速、方便的接口让我们迅速从 CMDB 中导入监控节点。

告警监控职能与告警处理职能常常是分割的，我们将所有的告警类型集中起来由专门的人员负责，告警添加人、告警监控人、告警处理人在一个不稳定、不灵活的监控平台下常常产生严重的误解。笔者目睹过恶劣的告警风暴的来临，以及风暴来临之后运维人员为了清理这些垃圾告警而做的重复而繁重的工作，要在一个受到 License 限制的界面上一条一条地删除海量的误报信息是何其无聊。告警处理人也会问告警监控人：“我已经在处理了，请不要通知我”但告警在 30 分钟后还是报出来了，到底要不要处理呀。告警监控人也会问告警添加维护人：“为什么你添加告警是如此不负责任啊，90% 都是误报、无须处理，如果真的有那么多特殊情况无须处理，则有没有办法提前注释，发版本停应用无须处理，大数据分析导致内存高无须处理，实在是太多原因无须处理了，面对“宁可误报一

千，不可错漏一个”的监控要求，运维人员一定要另辟蹊径，选择精细、精准化的监控管理方式。

### 14.1.2 监控平台建立的基础

在需要对某个对象进行监控时，我们可以选择不少于 10 种的方法，例如对 Linux CPU 负载的监控，不同的运维人员可能通过 `vmstat`、`top`、`sar` 命令来达到目的，为了快速地完成通知工作，我们会在脚本的末尾加上一句：`echo "`hostname` your cpu load is too high" | mail yuhe002@paic.com.cn` 来完成最后的告警通知工作，这是一种非常原始的监控手段，但对于一名基础 IT 运维人员来说，这样的一行脚本可能是他以后覆盖 IDC 所有监控的起始点，他会逐渐扩大代码量，渐渐规划自己完整的监控平台，比如建立标准文件夹结构，包括 `bin`、`conf`、`lib`，把所有需要的代码、配置文件、依赖库全部放进去，之后定义一些鲜为人知的规则、内创的 MVC、优雅的面向对象接口实现等。对于一名初级 IT 运维人员来说这些场景确实是一个比较好的锻炼机会，在一天或一个月内完成了自己对监控的理解。这样的监控实现方式也是程序员的“通病”，实现为王，能够跑起来再说，不关注问题的分析与设计过程。而实际上一套专业的面向企业级的监控平台是经过 3~5 年的运维经验后沉淀下来，并由一个团队来构建的。零散地将调度、采数、通知等动作积聚在一个脚本中的监控会让组织的运维工作越来越复杂。

再回头看看监控手段，我们可以通过哪些方式来获取监控对象的健康指标数据？监控与被监控对象是否在一个物理容器之内，监控主动方会通过 TCP/IP 去抓取所需要的数据，在网络设备中 SNMP 协议标准可供我们使用，而服务器操作系统对象则通常部署一套 Agent 在其上，服务器上的 Java 进程遵循 JMX 标准来抓取我们所需的内容。

行业内除了基础资源监控，还强调应用监控，模拟真实用户发起访问应用系统，并确认其是否可用，我们会通过 HTTPClient 等开源组件封装一个模拟业务的 HTTP 请求发起访问。应用监控告诉我们出问题了，但具体哪里出了问题还要由底层资源决定。端到端 E2E (end to end) 监控指的是覆盖应用所有组成资源各方面的监控，应用层是一个入口，在这个层面上的监控定制化需求更加强烈。

不要期望一个人、一个团队在半年内建立起一个完善的运维监控平台。它需要有丰富的监控经验来帮助我们明确监控需求与工具体系，通过需求来选择开源或者高性价比的商业化软件，同时监控工具要保持开放性，要么拥有 API 接口，要么有易读的源代码，运维团队通过开放性来提高监控管理效率，做好二次开发。

### 14.1.3 监控管理的WANT原则

好的工具让监控事半功倍，好的原则使告警一个不漏。IDC 的基础资源告警会集中起

来，将网络、服务器、应用等告警信息在一个终端输出。组织架构会设置专门的监控职能岗，7×24 对基础资源进行监控。所谓监控指监测与控制，器（工具）在于监测，术（原则）注重控制，再好的利器若未合理运用，就无法发挥其应有的作用。监控岗的原则是什么？记住四个词：Watching（监控）、Answer（响应）、Notify（通知）、Track（跟踪），总结起来就是 WANT。如图 14-1 所示。

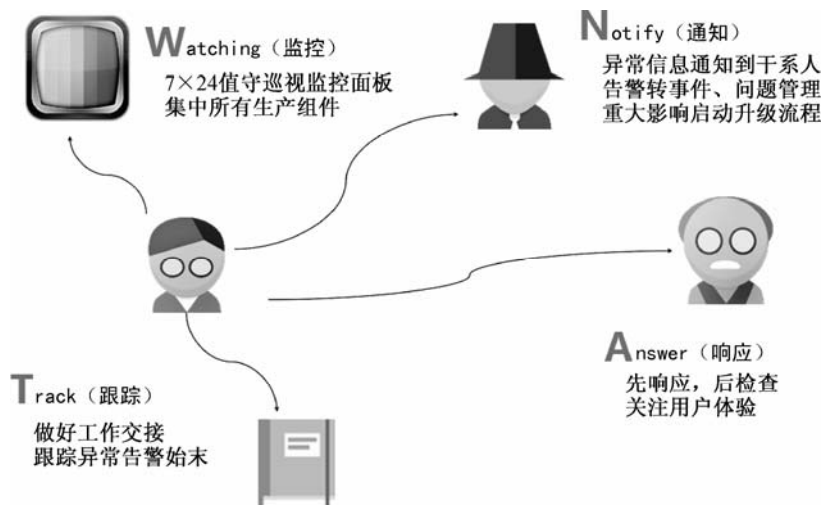


图 14-1 监控的 WANT 原则

**Watching（监控）。**这是 WANT 原则的最高级别，值班人员保证每隔 5 分钟扫一遍监控面板。有人会觉得这不够人性化，我们不是有手机吗？我们不是有 App 推送通知嘛？为什么非得一直盯着面板呢？对于几十万的监控项，常常会出现扑面而来、连连不断的告警，邮件、消息类通知方式除非采用分散式专人专属，否则极易形成垃圾风暴，实际经验证明：邮件、消息一般只能作为辅助监控手段。对于集中式监控管理方式，将网络、服务器、中间件等告警集中在一个面板，组织内会定义专门的监控职责，值班人员配备多屏，通过班次轮换，并行监控保证这个原则的执行。

**Answer（响应）。**对于紧急度高、影响范围大、级别严重的告警要第一时间响应。除了监控面板上显示的重要告警，组织内还会定义重大问题处理流程，监控值班人员会从不同的渠道收到告警通知，例如紧急处理流程的邮件列表、上级或相关部门的邮件等。值班人员收到邮件后都习惯性地先查问题再反馈，从内部看这种方式对问题处理的过程与结果无太大影响，但外部体验会很差。

**Notify（通知）。**将有效告警通知到处理人，重大问题及时升级。在集中监控下，监控和处理的职能并不在一个人身上，值班人员要对告警进行基本判断，这条告警是什么？是在发版本吗？是已知问题吗？过滤掉无效告警后通知到它的处理人。给处理人的通知分多种情况，有的通过监控平台关联事件系统，直接上报事件给后线工程师处理，有的则通过

CMDB 查到基础资源上的应用管理人员，邮件、电话告知处理。监控项与配置项是关联的，借助于监控平台上的“器”，将告警信息在面板上自动关联到监控组件的责任人，便于监控值班的同事查找。

对于关键服务和系统，或者大面积异常的告警，要及时升级，将问题与现象第一时间反馈给上级，待上级决策。

**Track（跟踪）。**记录下待确认、未解决的告警，做好班次交接。值班人员在班次内记录下所有待确认的告警，比如通知某技术线的人员查看某设备，但对方未确认告警组件已恢复，或告知无须关注的，这类告警要记录下来并转交给下一班次的人员，直到告警恢复，或告警已转入了问题、事件管理系统中，有专人负责处理，并有流程保证告警问题被解决。

## 14.2 对运维监控平台的需求分析

尽管已有了很多开源的监控平台工具供我们选择，例如 Nagios、Zabbix，但我们一开始先不进入一个具体的使用过程中，先搞清楚自己的原始需求是什么，需要一个什么样的监控平台，在明确自己要什么后再来选择合适的产品，会让我们后续的工作事半功倍。在附录中我们会对 Zabbix 的具体使用方法进行详细介绍。

### 14.2.1 一次监控过程，调度、规则、告警

在实际的生产环境中我们要对 WebLogic Java 进程进行监控，对 Linux 服务器进行监控，还要对网络设备进行监控。我们先来剖析 WebLogic 进程的监控过程。

监控程序按照一定的频率，遵循 JMX 标准向 WebLogic 进程发起请求，请求的内容是 WebLogic 的各项健康指标，JMX 的 MBean 名称和属性则是这个指标，每个 WebLogic 进程都有一批需要监控的指标，例如线程数、heap 大小等，在将指标数据取回来后，我们先进行数据存储，以便以后查看历史数据，之后按照一定的规则来判断是否需要发出告警。告警的类型又可以分为邮件、短信、监控面板。除了发出告警，你可能还需要对现场信息进行收集，打 threaddump 是经常使用的方法。我们用图 14-2 来描述我们这次监控的过程。

图 14-2 对一次简单的 WebLogic 进程的监控过程进行了说明，对于主机我们是否可以有同样的过程呢？监控程序发起对一个主机的监控访问，当然，主机会响应我们的请求吗？对于一般的网络 ping、telnet 操作会响应，但对于比较复杂的健康指标的请求会响应吗？比如当前打开的文件数，当前每个磁盘 I/O 的响应时间等，很可能不会，我们可以设计成 SSH 自动登录来将脚本嵌入命令行发送给 Linux 主机，也可以在主机上开发一个 Agent 程序来接收我们的请求，并按照我们的要求返回数据，不管怎样，我们也可以按照和 WebLogic 方式

一样请求我们想要的数 据，并且取回来后也可以存储到数据库中以 便于日后使用，数据的告警处理及告警判断之后的预定义行为设置和 WebLogic 监控是一样的。在确定主机和 WebLogic 进程一样可以对同一个模式进行监控后，我们试着提炼出参与这个过程的对象。

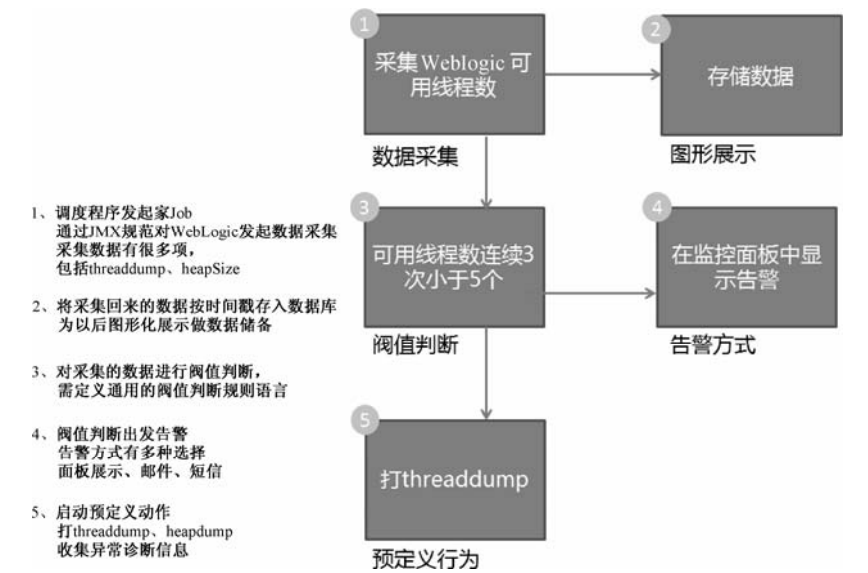


图 14-2 一次监控访问的过程

如图 14-3 所示是我们尝试着从 WebLogic 监控这个需求中提炼出来的类，我们对其逐一进行分析。

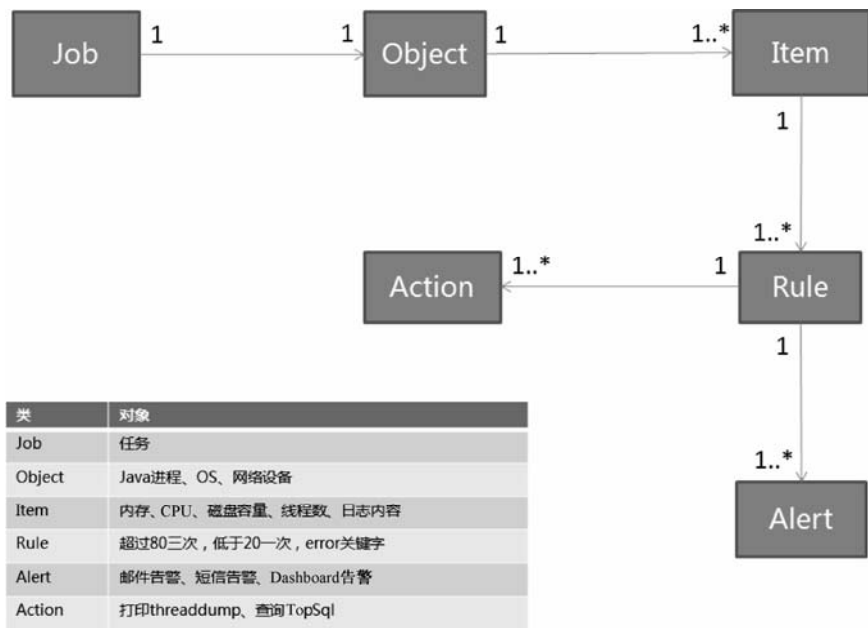


图 14-3 对监控对象的分析

我们在提到 Job 这个类时，最先想到的是发起一次监控访问，哦，不对，绝对不是一次。对于一个监控对象而言我们要持续地对其进行监控，并且遵循一定的时间周期，比如每隔 30 秒访问一次，既然出现了定时的周期性地完成一批不同类别的 Job，我们需要设计一个调度程序来完成这项任务。

对于每个 Job，我们周期性地发起监控请求操作，监控请求的处理时间需要包括在这个周期间隔内，我们的监控间隔是 10 秒，一个监控请求的响应是 2 秒，那么准确的动作应该是等待 8 秒后再开启下次监控情况，而不是在程序请求结束完毕后使用 `sleep(10)` 循环这个监控请求动作。有两种意外情况会发生：①调度程序自身繁忙而错过了对一个任务的启动，这种情况是可能发生的，因为调度程序要将 Job 分发给一个具体的执行线程处理，如果执行线程都在忙碌，则调度程序必须等待，一旦错过原来的调度时间，则调度程序应将任务启动时间调整为当下，下一次任务的启动时间以当下为基准来计算。②监控请求的过程可能会发生异常，监控请求的时间超过了监控周期，对于这种情况，调度程序也需要调整时间，避免重复启动程序，如图 14-4 所示。

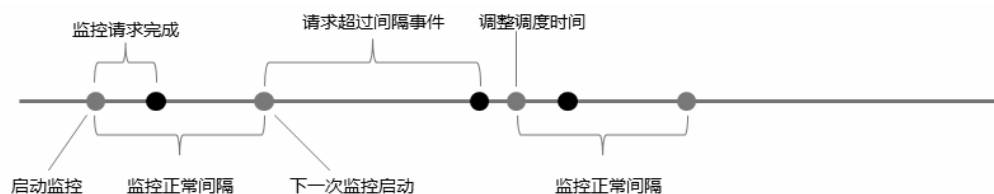


图 14-4 监控正常调整次序

除了在错过正常调度时间后进行自我调整，调度程序还要考虑到屏蔽时间，在设备维护、应用版本发布、定期批处理任务等时段监控获取的数值会发生异常，但这些时间段的异常是允许发生的，因此调度程序对每个任务要接收屏蔽时间段参数。

在 Job 的具体细节上，要一次性将所有的 Item 取回，例如获取 OS 信息，一次性将 CPU、内存、磁盘信息全部获取，而不是发起多次访问。另外，对于需要认证的步骤，只做一次，之后缓存凭证，以便下次使用。Java 的 JMX 访问如果每次都对远程用户进行认证，则执行效率会大大降低，因此需要缓存 subject 凭证信息。在设计调度程序时，对于周期性访问要区分第一次和以后每一次的逻辑，要留存一个空间保存 Job 的全局性数据。最初所有的 Job 是存放在数据库中的，第一次加载后要按照 Job 的目标访问地址进行错开排序，这样做的目的是避免产生对同一目标地址发起并发访问，因监控而造成目标对象的性能损耗。

根据以上论述我们需要一个调度程序（Scheduler），它依据一定的时间周期，以及屏蔽原则（Timer），对各种各样的任务（Job）进行调度。

有大量的开源调度库供我们使用，在 Java 领域我们会使用 quartz (<http://www.quartz-scheduler.org/>)，它可以集成到我们的应用中，按照计划的时间点及频度来执行任务。除了任务的调度与执行，quartz 从企业级架构上考虑设计问题，支持集群与事务。Object 为监控



对象。我们希望这个监控对象能够涵盖所有的监控实体，有服务、主机、网络设备。实体之间的差别在于如何访问、采集监控数据，这是一种动作。这些动作又基于不同的 LDAP、SNMP、HTTP 协议，或者不同的规范框架如 JMX。在这里需要定义一个足够抽象的接口来掩盖具体的动作细节，我们定义 poller 方法来采集监控对象的数据，采集的监控指标定义在 Item 中，因此 Item 会作为参数传入 collect 中。在采集数据时需要建立连接并考虑是否需要保持这个连接，在这里有一个连接管理过程。在 Job 中，对采集回来的数据是执行告警还是动作都是与 Object 关联的，Object 等同于任务中的门面对象，对采集回来数据进行告警，回调诊断方法都将在这个类中实现。Object 的实现类的复杂之处在于 poller 方法，以 JMX 来说，与远程对象建立连接的方式有很多种，笔者遇到过对 WebLogic 的不同版本通过 JMX 进行监控，发起连接的 client 类名一致，但版本不同，最后在建立连接的动作中实现不同的类加载器才解决这个问题。开源产品可以提供一个监控处理的方法与框架，但在细节层面上很难有一款产品的灵活性足够满足我们的需求。对于运维来说，安装、配置、启动一个开源产品非常容易，但要驯服它，完全按照我们的思路运转则需要花费大量的精力。不管是开源产品还是商用产品，只有公开源代码并且具有对它足够的控制能力后才能自由地支撑起 IT 运维，对一个开源产品的二次开发需要专业的开发人员，在此强调了运维人员的开发能力。

Item 为监控项，就是监控对象自身的监控指标，每个 Item 有一个独有的名字，比如 WebLogic 线程数，我们会将它取名为 WebLogic.10.jmx.thread.active.count，对这个名称加上别名“WebLogic10 活动线程数”，我们还要定义监控项的值类型，是浮点、整型还是字符型。对于获取的值我们将如何判断并处理，因此必然关联上 Rule。Item 还要携带一类有效数据来帮助 Object collect，如果是 JMX，则它可能是 MBean name，而对于 SNMP，则是 oid 编码。我们要预留足够的空间给有效数据，它区分为静态数据与动态数据，我们以一个磁盘检测的 Item 为例，Item 静态的是 DiskUsageStats，即磁盘使用率，它告诉 Object 对象在收集时获取什么类型的数据。动态名称则是一个具体的磁盘名称，例如/nfsc/im\_core5，告诉 Object 具体采集哪个磁盘的数据，设计 Item 时要考虑到各种问题发生的情况，有时动态数据还是通过正则表达式实现的，比如/nfsc/im\_core\*，要检索所有符合正则表达式的磁盘。

Rule 为规则，对 Item 的值进行判断，判断是否要发出一次告警或动作，并且根据 Item 的值解析完毕后设置一个状态，状态包括 OK、INFO、WARN、CRITICAL 等。对于 Item 来说这是它当前的状态，对于 alert 来说这是告警的级别，一个 Item 关联到了多个 Rule，那么以最高级别的状态为准。在 Rule 的属性中最重要的是规则表达式，“在阈值超过 90，并连续三次后，确认为 WARN 状态”这就是一个规则语言，其中包括了逻辑，规则语言的重要性在于将逻辑作为参数传递，将逻辑动态化了，Rule 如同规则引擎。我们来看看关于 WebLogic 排队线程数的规则表达式，“critical={>,0,2}”，这句话表示当数值大于 0 且连续两次后则将状态修改为 critical，即严重。

我们看看 Zabbix 的规则表达式，下面是一条 CPU 负载的表达式：

```
{www.zabbix.com:system.cpu.load[all,avg1].last(0)}>5
```

“www.zabbix.com”表示监控对象，这里的“system.cpu.load[all,avg1]”类似于 Item 的唯一属性，last 表示取最近的一次值，当最近的一次值的负载大于 5 时，则认为有问题。Zabbix 的规则比之前的要复杂也完善很多，再来看下面这条规则：

```
{www.zabbix.com:system.cpu.load[all,avg1].last(0)}>5|{www.zabbix.com:system.cpu.load[all,avg1].min(10m)}>2
```

当 CPU 的负载大于 5 或者在最近 10 分钟内负载大于 2 时确定为异常。我们注意到，Zabbix 的设计与前面不一样了，它的 Rule 与 Item 之间是没有关系的，Rule 以一个集合的方式直接关联到 Object，这样才可以实现对不同的 Item 之间的与、或的规则定义。Rule 到底和谁关联更合适？显而易见，Rule 应该与 Object 关联才能在全量的 Item 数据中进行匹配判断，这才是最灵活的，相比于在做完一件事后判断其对错，更合适的方式是在做完所有的事情后再全局判断，设计细节会给产品的扩展性带来很大的影响，需求永远在变，为了保持绝对的灵活性，必须深挖需求，这样才能辨识对象之间到底是否存在必然的联系。我们在之前的抽象关联图上再进一步细化，并调整关联关系，如图 14-5 所示。

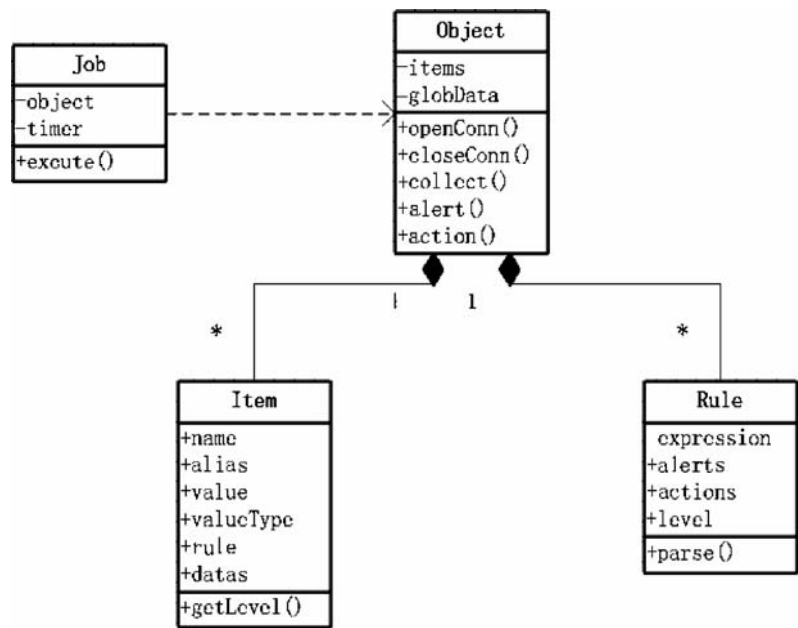


图 14-5 监控对象的关联设计

在规则表达式中会涵盖很多元素：阈值，最终 Item 值对比的数值；操作符，大于、等于或小于；重复次数。Zabbix 还定义了一些函数对值进行操作，比如取值时间段等。

Alert 为告警。在 Rule 命中后则会进入 Alert。告警的方式很多，发短信、发邮件、dashboard 面板展示及其他方方面面。对于 Rule 触发的 Alert 而言，它仅仅把一条消息发送

给告警的接收者，但要注意的是在生产环境中非常容易出现告警风暴，经验证明将告警邮件、短信广播到所有的当事人，不如将告警集中在一个 Dashboard 上，由 7×24 的值班人员负责值班监控。在 Dashboard 面板上，值班人员对告警的响应方式便捷很多，他们可以采用注释、确认、转事件、升级、问题管理等诸多方法，一个注释的告警要说明注释原因，以及注释时间段。为了防止出现风暴、冗余现象，我们要做一层 filter，而这层 filter 如同 Rule 一样也是基于规则的，它们会关联到 Object 的维护时间、变更窗口，甚至与实际的应用运维部署动作关联，从而大量减少了无效告警。在 Alert 传递给终端用户前必须经过层层过滤，从告警类型与告警过滤上来看，开源产品本身无法满足要求。

Dashboard 告警和 MSG（短信、邮件）的告警方式各有所长。Dashboard 是主动型的，由监控人主动从面板中“拉”回告警信息。在运维职责中，有专门的值班人员负责盯着监控面板，面板定期刷新，将最新的告警信息更新进来。MSG 是被动型的，告警信息会推送到接收人的终端设备上。Dashboard 的优势在于在一个集中的面板上展示了所有的告警信息，便于关联分析；对于重复的告警只显示一条，可防止告警风暴；对于告警的处理可关联 CMDB，监控人员判断是否需要联系上层业务应用管理员；对于关联事件管理系统，对于已知的问题要上报给后端事件团队。而以 MSG 方式，在理想情况下告警处理人员可以做自己的事情，只有当告警通知时才开始处理。大型 IDC 运维中心一般将 MSG 作为一种可选的通知方式，而重点采用 Dashboard 方式。智能手机已经成为日常生活中必不可少的随身件，将 Dashboard 传统的网页版移植到 App 上，既可以保持原有优势，也可以得到 MSG 通知功能。

由于 Dashboard 是所有告警信息的集中面板，所以必须在上游建立告警阈值规则时就确保设置精准，否则会出现大批无须处理的无效告警，这种规则的调整是一个长期的过程，监控项与配置项一样有专属的 Owner 团队，接收前端监控人员的反馈，及时消除无效规则。上述的反馈过程不是马上见效的，为了保证面板整洁，还需要提供诸多的功能帮助监控人员。告警注释功能就是其一，注释告警时定义注释时长、注释原因、注释人；告警转事件，与 ITIL 流程管理系统关联是功能的第 2 个，将一条告警转发给后端团队处理。如图 14-6 所示。

1 注释告警  
2 转报事件  
3 邮件通知

主机名	主机 IP	级别	应用分级	消息	时间	监控图	主机 OS	上层告警	处理
CNSH211355	10.37.232.151	●		App logmonitor_middleware = no return value	08-27 09:43:30		WINDOWS-2008EN	SLFNY02	
CNSH211355	10.37.232.151	●		App logmonitor_os = no return value	08-27 09:43:30		WINDOWS-2008EN	SLFNY02	
CNSH211355	10.37.232.151	●		App trans_file = no return value	08-27 09:43:30		WINDOWS-2008EN	SLFNY02	
CNSZ022947	10.33.96.149	●		Cpu sys = 73.57238	08-27 10:27:47		WINDOWS-2008CN	CNSZ060229	
CNSZ022947	10.33.96.149	●		Cpu used = 100.0	08-27 10:25:46		WINDOWS-2008CN	CNSZ060229	
CNSH022255	192.168.193.198	●		/D: disk space used 99.9648410044611%	08-27 10:25:36		WINDOWS	CNSH060284	
CNSZ031332	10.33.88.13	●	一类	Log os = Has_log_alert	08-27 10:03:48		RHEL-5.X	99D0747	
CNSH022192	10.37.84.64	●		Cpu used = 100.0	08-27 10:02:34		WINDOWS	CNSH060126	

图 14-6 告警面板的无效清理

在 Dashboard 上要保证信息精简有效，拒绝无效的数据与格式，类似于在面板中嵌入一个仪表盘的功能完全没有必要，占用空间大，且只能反映一条当前容量信息，这类功能属于“展示”类需求，在监控运维中没有实际用处。如图 14-7 所示。



图 14-7 仪表盘在监控告警上的浮华

Action 为动作。严格地说 Alert 属于 Action 的一类，那为什么要加以区分？Action 是一个一个回调函数，当 Rule 命中时发出我们要采取的行动。我们常常苦恼于问题在刹那间溜走，虽然保留了一些痕迹、线索，但这些内容是不足以让我们找到问题根本原因，我们希望有一个自定义的接口让我们在第一时间运行诊断工具，比如 DB 主机 CPU 高，但这个告警出现时，我们可以定义一套诊断序列来分析哪个进程 CPU 高，如果是 DB 进程，则我们看看等待事件、看看 topSQL 等。对于 Java 进程，它会打出 threaddump、heapdump，对于 heapdump 的分析还可以延伸至对象之间的引用关系，看出真正是哪个对象的占比高。提供一个开放的接口会引发大家的思考，进而朝运维自动化迈进。

在核心需求的分析完毕后，我们可以对此模块展开设计工作，同步进行的还有对其他需求的收集与分析，敏捷迭代开发的过程在于从需求中选择最核心的部分进行分析、设计，并开始编码实现，在随后的迭代过程组中继续完善其他非核心需求。

### 14.2.2 数据图形化：百分位裁剪、趋势分析、正态分布

核心需求所关注的是 Object 的可用性、健康与否，以及通知监控人员处理。所有的监控系统同时需要具备将存储的 Item 监控项指标数据图形化展示的能力。图形化数据体现在容量、性能两方面。容量与性能的区别在于我们是否对某一项监控指标有了明确的基线，并且能够换算成百分比。磁盘的空间使用率、CPU 的系统空间使用率就是已度量好基线的容量。而性能则不同，它的基线不是显而易见的，它代表着在单位容量的情况下完成的业务请求、响应时间、吞吐量，性能调优可以帮助我们在固定的容量资源下完成更多更好的任务，当一切趋于平衡时我们要度量出单位容量完成的业务处理能力。图形展示中的这些容量、性能数据对我们有什么帮助？我们会在以下情况中使用这些以时间轴为单位在图形

中展示的数据。

### 1. 判断异常点的依据

我们期望大多数异常的线索在日志中体现，但很可惜，异常点本身就不是以一个可记录的 **Error** 出现的。有些异常会在没有任何日志记录的情况下间歇性出现，这时我们唯一能够做的就是从历史 **Item** 数据的突变点中查找线索。将数据按时间轴排列在曲线图、柱状图中是我们常用的展示数据的方式。采集到的大量数据在异常判断时往往只会用到一点点，即便异常本身不出现，某些 **Item** 数据在图中的波动也会引发相关人员的关注并对问题进行查找。由于采集的数据量比较大，而一般关注的数据的时间都是最近点，在数据点的存储上会将 7 天或 30 天后的数据粒度粗化，以释放数据存储空间。每天晚上会有一个后台批量程序进行数据粒度的有效裁剪。

百分位数法（percentile）是数据裁剪的常用算法。它是统计学术语，如果将一组数据从小到大排序，并计算相应的累计百分位，则某一百分位所对应数据的值就叫作这一百分位的百分位数。可表示为：一组  $n$  个观测值按数值大小排列，处于  $p\%$  位置的值叫作第  $p$  百分位数。

我们常常说 **P97**，举例而言，采集回来的 CPU 使用率数据是以 30s 为间隔的，在 100 个数据中，我们对其进行排序，保留排在第 97 位的一个数据，其余的 99 个值全部丢弃。在实际情况中我们不会严格地按照 **P97** 来计算，一般会采用 1:7 的缩放比例，7 个数据排序中取第 6 位作为有效位，从而压缩空间。

对于某些特殊的 **Item** 指标的监控数据中不高的值，比如空闲线程数，数量越少代表越要关注，因此采样时要使用 **P3**。

### 2. 定义基线

我们需要定义 **Item** 指标数据的基线，某个核心业务操作的数量在什么范围内是正常的，在什么情况下需要引起我们的关注。在业务核心功能访问量上常常会试着将采集到的数据进行基线建立。比如在寿险出单系统中，我们采集了过去三个月每周一早上 10:00~12:00 的出单数据，将其分布在一个  $X$ 、 $Y$  数据轴上， $X$  是我们的录单数，将其作为一个随机变量，按照正态分布模型，我们可以计算出当前哪些数值在异常范围内，且需要我们关注。

正态分布（Normal Distribution）是自然科学与行为科学中的定量现象的一个便捷模型，也叫作钟形分布，这个名字是因为正态分布的数值在图形上类似一口钟而得来的。在一系列的数值当中，中值附近的数值数量最多，而偏离中值的数值数量则不断减少。人类社会的很多行为都符合正态分布的特点，那些“大多数”都集中在中值附近，而“非主流”则偏离中值。如图 14-8 所示。

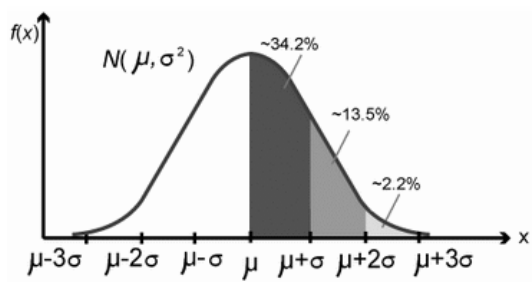


图 14-8 数据分析上的正态分布

3. 趋势分析

什么时候会出现瓶颈，什么时候要追加资源，对于那些已经给出了预警峰值的容量型数据，可以通过趋势图推测出未来到达峰值的时间点，提前做好扩容准备工作。

最小二乘法（Least squares）是一种数学优化技术。它在 X、Y 轴上采集一定数量的点，通过最小化误差的平方来计算趋势直线。对每个 Item 评估出其预警峰值，在与趋势直线相交的点为瓶颈点。如图 14-9 所示。

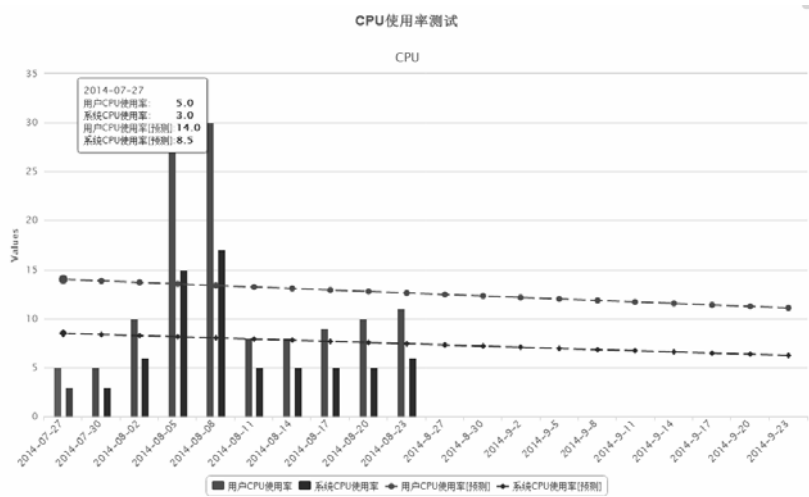


图 14-9 数据分析上的趋势分析

图形展示模块是可配置的，可体现在一张图中，我们可以将任意 Object 的 Item 组合在一张图上按时间点一起展示。对于多张图表，我们也可以自定义地将它们组合在一个页面上，从而形成由用户完全自定义配置的监控对象集合。

有很多开源的 Web 前端图表工具可供使用，按照 API 要求将数据加载到图表中。在选择通用开源工具包前要考虑几点，包括功能上是否完全满足自己的需求，开源社区的资料是否充足、在浏览器的兼容性上是否处理得很好。

Highcharts (<http://www.highcharts.com>) 是一款纯 JavaScript 语言编写的图表库，我们能

够使用它很便捷地在 Web 网站或 Web 应用中添加交互性的图表。在兼容性上它做得非常出色，几乎支持所有的现代浏览器，包括 IE 6+、iPhone、iPad、Android。它在标准（W3C 标准）浏览器中使用 SVG 技术渲染图形，在遗留的 IE 浏览器中使用 VML 技术来绘图，完全基于本地浏览器技术，不依赖任何插件（例如 Flash、Java），不需要安装任何服务器环境或动态语言库支持，只需要两个 JS 文件就可以运行。目前可绘制直线图、曲线图、面积图、曲线面积图、面积范围图、曲线面积范围图、柱状图、柱状范围图、条形图、饼图、散点图、箱线图、气泡图、误差线图、瀑布图、雷达图等，其中很多图表可以集成在同一个图形中形成综合图，并支持图表的缩放。

Highcharts 具备足够的灵活性，提供丰富的 API 接口，可以很方便地对图表的任意点、线和文字等进行增加、删除和修改，支持众多 JavaScript 事件，其结合 jQuery、MooTools、Prototype 等 JavaScript 框架提供的 Ajax 接口，可以实时地从服务器取得数据并实时刷新图表，支持多种数据形式，包括 JavaScript 数组、JSON 文件、JSON 对象和表格数据等，这些数据来源可以是本地的不同页面，甚至是不同的网站。

Highcharts 在国内拥有开源的中文站，到目前为止该网站拥有“中文论坛”“在线演示”“中文教程”“中文 API”“在线测试”“相关资源”六大核心资源，其中的资源还在不断完善。

监控项与配置项一样，要录入到监控平台系统中管理，在监控平台上也要考虑用户界面的基本功能，包括数据录入、权限控制，这与配置管理的要求是样的。Monitor Object 和 CI 的属性在很大程度上是重合的，监控项依赖配置项，二者的数据要打通、共享。

### 14.2.3 开源的借鉴与选择：Zabbix和Nagios

在选择监控平台系统时，根据组织综合情况可以选择开源、商业的解决方案。一套完善的监控系统需要几年的打磨，平安科技用了 5 年，经历了从商用解决方案到自开发工具的过程，在监控平台工具上积累了丰富的经验，建立了专门的 ITIL 工具开发、运维团队。没有实践经验的团队应当选择产品化的开源、商业化产品起步，渐进式地演化监控平台产品，最终形成适合企业自身的产品工具。

对于中小企业的监控我们选择开源产品帮助我们起步，在选择开源产品时要从一些角度做调研工作。

（1）活跃程度。指开源产品是否足够活跃，开发人员对该项目、产品的投入是否频繁，开源产品在设计之初是否遵循了既定的标准以保证更多的人投入到开发之中，开源产品的第三方支持、插件是否丰富。如果你发现一个产品已经有半年以上时间没有更新，则该项目很可能已经处于退役阶段。

（2）许可证。如果你所使用的产品将投入到商业化运作中，则请特别留意开源产品的

License 限制，在尊重原作者的知识产权的同时，也遵守相关法律条约。之前提到的 Highcharts 工具针对个人用户及非商业用途免费，并提供源代码下载，我们可以任意地修改它。单商业用途需要购买。

（3）文档。是否提供简洁明了的用户手册，是否有足够多的中文文档，以及博客、实践经验供我们参考。

（4）成本。开源产品也有成本？开源产品并不意味着免费，在选择一个开源产品后我们将投入极高的学习成本、安装配置成本、运维成本，对于一个功能完整但用户界面无比糟糕的产品，你将投入相当多的维护成本。

（5）功能。你所选择的产品是否能够满足你的需求，在你“驯服”它的同时能够获得你所需要的内容。

在社区里比较热门的开源监控平台软件有 Zabbix、Nagios，二者都是优秀的作品，有足够的活跃度，有一大群人员支持并保持不断更新。在国内已有相当多的用户，中文文档、教程资源很丰富，感谢社区的贡献者们，这一节我们主要从功能的角度评价这两个开源产品，帮助大家借鉴与选择。

Zabbix 的创始人 Alexei Vladishev 于 1995 年在一家荷兰银行工作，监控平台的需求就是从这家银行开始的，随后其一直专攻监控软件开发，于 2001 年将 Zabbix 贡献给社区，宣布开源，但直到三年才发布其第一个稳定版本，Zabbix 的发展与 Nagios 不一样，其最初的目标是以商业软件的形式发布的，其分析与设计过程是基于大型金融企业 IDC 的监控需求进行的，因此具备一定的封闭性，但反过来保证了其风格的一致。虽然 Zabbix 已经开源，但其更像一个完整的产品，目前支持 MySQL、PostgreSQL、SQLite、Oracle 等数据库，后端采用 C 语言编写，前端基于 PHP 语言。

Nagios 比 Zabbix 在开源社区更具有影响力，其于 1999 年就发布了第一个版本，随后得到了无数系统管理员粉丝的支持，当时 Nagios 和 Cacit 的整合成为了标准的开源监控解决方案。Nagios 最初是由系统管理员开发出来的，核心是基于配置文件与通信标准的调度中心，其重心并没有放在具体的监控实现上，我们可以看见很多实现需要依赖插件和 addon，这种足够的灵活性是 Nagios 的设计初衷，正由于其灵活性及长期以来的影响力，社区发展出了很多基于 Nagios 的变体，包括在用户界面、数据存储、报表信息等功能上的补充。可以说 Nagios 在使用上是丰富多彩的，却失去了标准。

随着数据中心的监控对象明确（硬件、网络、操作系统、服务），监控通信协议的统一（SNMP、JMX、agent），监控需求与部署架构最终发展到相同的方向，正所谓条条大道通罗马，让我们一起看看这通用的架构，并逐一点评。

### 1. 调度模块

Nagios 的监控对象（Object）分为 host 与 service，host 指监控的服务器，service 指在服



务器上运行的服务，例如一台 Linux 的主机是 host，则在 Linux 上运行的 HTTP 服务、SSH 服务等均是 service。下面是监控节点的配置。

#### 清单一. host 配置文件:

```
define host{
    use                Windows-server
    host_name           winserver
    alias               My Windows Server
    address              192.168.1.2    ; Windows 服务器的地址
    check_period         24x7
    check_interval      5                ; 调度 host 正常检查的时间间隔为
5 分钟
    retry_interval      1                ; 调度 host 在异常出现后重试的间
隔为 1 分钟
    max_check_attempts 10
    check_command        check-host-alive    ; host 监控命令的具体实现
    notification_period  24x7
    notification_interval 30
    notification_options d,r
    contact_groups       admins
}
```

在 host 配置中注明了 host 名称及 host 地址。

#### 清单二. service 配置文件:

```
define service{
    use                generic-service
    host_name           winserver
    service_description NSClient++ Version
    check_command        check_nt!CLIENTVERSION ;服务监控命令的具体实现
}
```

Service 依赖于 host 配置，与调度相关的信息是直接填写在 host、service 配置文件中的，包括监控频率 check\_interval、监控重试次数 retry\_interval。这些 host、service 是通过什么方式去监控的呢？其开放命令行接口，由外部插件实现。配置文件的 check\_command 项引用了外部具体的可执行文件。

#### 清单三. command 配置文件:

```
define command{
    command_name        check-host-alive
    command_line         $USER1$/check_ping -H $HOSTADDRESS$ -w 3000.0,80%
-c 5000.0,100% -p 5
}
```

在需求分析单元我们谈到了具体的监控项（Item），Nagios 模糊了 Item 的概念，具体监控什么指标 Nagios Core 并不关注，它不会关心你到底是监控 CPU 负载还是内存，它将这部

分内容交由外部插件来实现，最初的方式是独立的可执行文件。Nagios Core 只负责调度并执行这些外部插件。

模糊了 Item，也就是将数据采集与规则告警的职能全部委派给了插件。只需要了解一些简单的指导原则，就可以轻松地编写你自己的插件，Nagios 在每次查询一个服务的状态时，会产生一个子进程，并且它使用来自该命令的输出和退出代码来确定具体的状态。退出状态代码的含义如下所示。

- OK: 退出代码，0 表示服务正常工作。
- WARNING: 退出代码，1 表示服务处于警告状态。
- CRITICAL: 退出代码，2 表示服务处于危险状态。
- UNKNOWN: 退出代码，3 表示服务处于未知状态。

最后一种状态通常表示该插件无法确定服务的状态。例如，可能出现了内部错误。

下面提供了一个 Python 示例脚本，用于检查 Linux OS 的平均负载。它假定 2.0 以上的级别表示告警状态，而 5.0 以上的级别表示危险状态。该数字代表操作系统的进程队列中等待获取 CPU 执行权的进程数，这些值都采用了硬编码的方式，并且始终使用最近一分钟的平均负载。

### 清单 4. Python 脚本文件：

```
#!/usr/bin/env Python
import os,sys
(d1, d2, d3) = os.getloadavg()
if d1 >= 5.0:
    print "GETLOADAVG CRITICAL: Load average is %.2f" % (d1)
    sys.exit(2)
elif d1 >= 2.0:
    print "GETLOADAVG WARNING: Load average is %.2f" % (d1)
    sys.exit(1)
else:
    print "GETLOADAVG OK: Load average is %.2f" % (d1)
    sys.exit(0)
```

### 清单 5. 定义新插件：

```
define command{
    command_name    check_load_avg
    command_line    /path/to/check_load_avg
}
```

Nagios 插件作为独立的可执行文件被 Nagios Core 调用，在灵活性方面，与 Nagios core 之间有输入、输出的数据传递关系，解除了纯粹的硬编码方式。在配置文件中可定义从 Core 传递到插件的参数，插件也可以将数据分列输出给 Core，从而存储到历史数据库中。

Nagios 的监控实现交给了插件，每次调用都启动一个新的执行流，加载外部独立的可执行文件，这将导致调度程序的执行性能大大降低。对于同样的监控对象数量，Nagios 所要求的硬件资源远高于 Zabbix。从这里也可以看出 Nagios 的架构并不以海量监控项、产品化为目标。规则与数据采集混合在了插件中，为实现一个监控目标而一次性输入，并未考虑到以后规则的变化是基于采集数据的，看似灵活却并不灵活。Nagis 是从系统管理员角度出发的，通过开放命令行接口、自定义执行程序来“黏合”成一套监控平台。

Zabbix 在调度上定义了两个对象：Host 与 Item。Host 是要监控的一个设备，如图 14-10 所示。它可能是一个交换机、服务器或者进程，而 Item 是从 Host 上收集的一条数据。Zabbix 的 Item 分为很多类别，比如 Agent、SNMP、JMX，这些类别代表了数据采集的具体方法，Zabbix 不采用自定义命令行、独立执行程序的监控方式。例如它不会让你关联自己的 Ping 命令脚本来实现存活性监控，而是采用它内部定义的 Agent 类型的 Item，并输入“agent.ping”的 key 值实现。Zabbix 的监控逻辑代码与调度程序集成在了一个进程之中，除了 SNMP 的底层库，不再依赖于任何第三方工具，其性能要远高于“原生态”的 Nagios。

图 14-10 Zabbix Host 定义

Zabbix 的作者在被问及与 Nagios 的比较时，常会“谦虚”地说：“我们是从设计草图开始的”言外之意就是 Zabbix 一开始是面向产品化方向发展的，其在系统对象分析与设计上要比 Nagios 高明。Item 仅代表要采集的数据条目，至于如何触发通知与告警则依赖于另一个 trigger 对象，这与 rule 有相同的作用，trigger 的语法规则强大，不仅可以根根据采集的数据值进行判断，还可以追溯一段时间内的数值变化，并且允许多个条件组合。

## 2. 监控访问

调度程序按时间与频度定期地发起监控访问，调度程序监控者与被监控对象一般分布不同的网络节点上，发起监控访问时需要考虑两点：①发起访问的具体协议；②被监控

对象上是否接收此协议。大多数被监控对象都支持特定的访问协议，并且会做出响应，例如网络设备上的 SNMP，JVM 进程的 JMX，这类监控对象不需要额外部署 Agent 程序。而对于操作系统而言，虽然它也支持通用协议，但是由于平台类别太多，并且操作系统层面的监控项内容更加丰富、灵活，所以无法在一种通用协议上完整地表述我们的所有需求。面对这种情况，监控平台往往会在各类操作系统上开发一个 Agent，由它负责接收、响应调度程序监控者的监控访问请求。在这里我们称监控者为 Server，称被监控者为 Agent。

Server 与 Agent 之间的交互方式有两种，这两种方式的区别在是谁主动发起了访问。从 Server 的角度看，我们称这两种方式分别为 poll、accept。poll 主动发起访问，accept 则被动接收请求。

Nagios Core 的外围有一批附属项目（Addon Project），当前的数量已经达到上百个，它们对 Nagios Core 的功能进行扩展，Nagios 的外部监控功能由几个项目实现。

在远程被监控 Linux / UNIX 的机器上部署了 NRPE 的 Agent，Nagios Server 通过本地的 check\_nrpe 发起对远程主机的监控，在被监控机器上执行监控任务还由插件完成，NRPE 只是一个命令转发者。如图 14-11 所示。

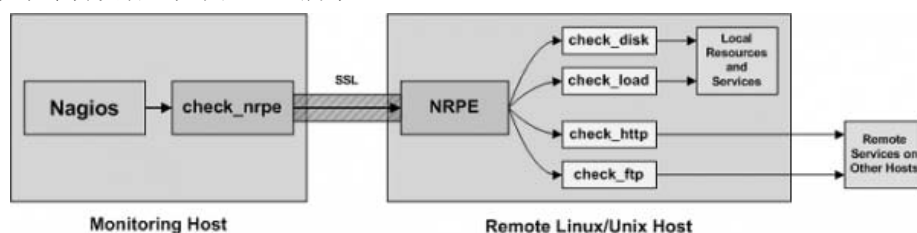


图 14-11 NRPE 概要图

NRPE 由以下两部分组成。

- check\_nrpe 插件：位于 Server 上，通过它发起对 Agent 的访问。
- NRPE daemon：运行在远程的 Linux 主机上，即 Agent。

整个的监控过程如图 14-12 所示。

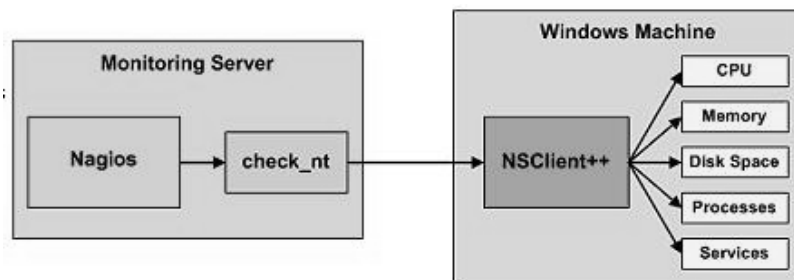


图 14-12 NSClient 对 Windows 操作系统的访问

在 Nagios 需要监控某个远程 Linux 主机的服务或者资源情况时：

- (1) Nagios 会运行 check\_nrpe 插件，告诉它要检查什么；
- (2) check\_nrpe 插件连接远程的 NRPE Daemon，的方式是 SSL
- (3) NRPE Daemon 会运行相应的 Nagios 插件来执行检查；
- (4) NRPE Daemon 将检查的结果返回给 check\_nrpe 插件，该插件将其递交给 Nagios 做处理。

NRPE Daemon 需要将 Nagios 插件安装在远程的 Linux 主机上，否则，daemon 不能进行任何的监控。

对于 Windows 主机，Nagios 采用 NSClient 进行监控，其部署架构与 NRPE 类似。唯一的不同是 NRPE 在远程被监控主机上还是依赖插件，而 NSClient 将所有执行逻辑放在了单个进程中。

NRPE、NSClient 属于主动发起访问的方式，在服务端其每次仍旧是 Fork 一个新进程，加载独立的可执行程序运行。另外每次调用都是同步方式，当监控对象过多时，调度队列中的任务会因为等待前面的任务而出现拥塞，从而错过准确的调度时间。这两个因素都会导致性能大大降低。

NRDP (Nagios Remote Data processor)

Nagios 的另一个附加项目在服务端部署了一个 Accpetor，专门接收被监控节点传来的数据。

- (1) 远程 Client 通过 NRDP Agent 向 NRDP 服务器提交请求。请求中的数据包括：已授权的有效令牌、Host 和 Service 名称、监控的输出与状态、NRDP 服务器执行的命令。
- (2) NRDP 验证客户端的请求令牌。
- (3) 依据用户提交的数据调用相应的插件进行处理。
- (4) NRDP 返回结果信息的 XML 格式给客户。

整个处理过程如图 14-13 所示。

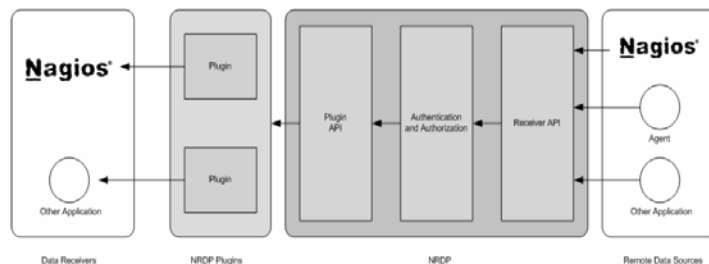


图 14-13 NRDP 处理过程

NRDP 是一个遵循 HTTP 的 Acceptor，它接收来自客户端的请求，实际上将监控的任务完全分配到客户端处理，处理完后将状态、输出发送回服务端。比较简陋的是其开源版本只提供了一个 PHP 页面的客户端 Sender，而在商用版本中提供了专门的客户端 NRDS，添加了配置同步功能。如图 14-14 所示。

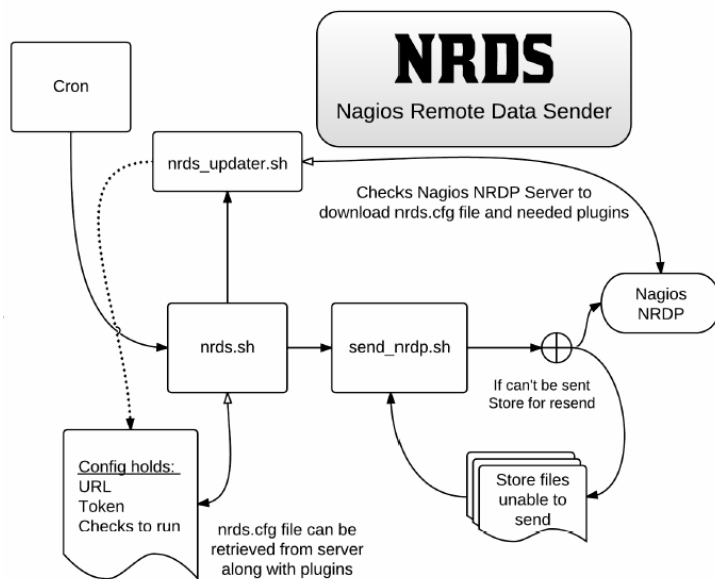


图 14-14 商用版本的客户端 NRDS

NRDP 的 Passive 被动方式一定能够释放服务端的监控压力，不用频繁地 Fork 进程并同步等待监控对象的返回。但在海量监控项下，我们需要进一步考虑服务器的负载分流。解决此类问题的通用方式是按照一定规则将负载分布式地发散到不同的 Server 上，保留唯一的中央节点。Host Group 将监控负载分配给在不同机器上运行的调度器，称之为 Proxy，由它完成其所管辖范围内的监控项的调度、访问工作，将处理结果一次性地返回给中央节点。

要做到主动发起访问的监控无等待超时，则必须将同步方式转变为异步方式。将数据的交互存放在消息队列中，生产者与消费者通过队列解耦。

Nagios 性能的提升还可以将调度的 Fork 生成进程方式修改为动态链接库，在 Server 启动、第一次调用时就将所有监控逻辑加载到进程内存空间中，而不用每次不停地创建、分配、注销进程。

### 3. NEB (Nagios Event Broker)

NEB Nagios Core 提供了一种事件机制，允许外部程序通过动态链接库加载到 Nagios Core 进程中，监听具体的某一系列事件而产生特定动作。

NEB 使外部动态链接库代码逻辑获取到 Nagios 的内部信息。当 Nagios 启动时，NEB 模

块 HOOK 到 Nagios 的核心进程。NEB 模块使用回调函数，当特定事件发生时，就执行这些回调函数。

在原生态的 Nagios Core 基础上可以有很多第三方项目利用 NEB 机制对监控架构进行重新设计，其中比较著名有 ndoutils、mod\_gearman。

Mod—Gearman 降低了 Nagios core 的负载，在 Server 端它的嵌入是遵循 NEB 机制的。在任务派发上它采用了异步方式，将监控任务派发到不同的消息队列中，由不同 Worker 进行处理。如图 14-15 所示。

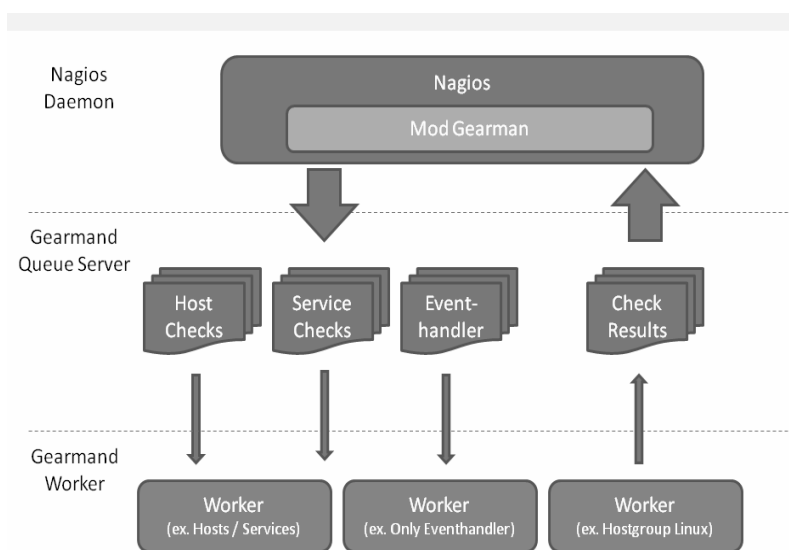


图 14-15 Mod—Gearman 动态链接库的 NEB 机制，消息队列接口 Server 和 Agent

Mod—Gearman 的工作流很简单：

- (1) Nagios Core 执行监控调用检查；
- (2) Mod—Gearman 的 NEB 模块截获这类事件；
- (3) Mod—Gearman 将事件作为一个任务放到队列中；
- (4) 一个 Worker 进程获取一个任务，发起监控调用检查，并把结果放回到 check\_results 队列中；
- (5) Mod—Gearman 从 check\_results 中获取数据，交由 Nagios Core 处理。

Nagios 的发展可以用“无组织、无纪律”来形容，由 Core 衍生出来的第三方项目越来越多，但官方从来没有努力将这些项目规范统一、集成打包发布，做到简单地开箱即用，导致每次出现一个新组件，我们不仅要重新学习其配置方式，还要逐一解决新组件产生的问题，用起来很头疼，这可能与 Nagios 的商业计划有关系，官网上有商业版本的 Nagios

XI，以至于开源版本简陋而零散。

开源社区自有其办法，也有人看到了同样的问题。2010 年 7 月，Mathias Kettner 将当时常用的 Nagios 插件与附加项目集成打包，新成立了 OMD 项目，全称为 Open Monitoring Distribution，它是一个围绕 Nagios Core 构建的分布式开源监控集。在 Nagios 基础上融合了各类第三方插件，以实现一个完整的高性能的可视化的分权限管理的监控系统。项目主页是 <http://omdistro.org>，其提供了 RH、Debian、Suse 和 SRC 等各种安装模式。

Zabbix 与 Nagios 截然相反，它并没有太多的插件与第三方附加项目，它的监控访问完全在独立的进程中完成，几乎未依赖外部可执行文件，这在性能上是一个不小的提升。Zabbix 在各类操作系统平台上实现了自己的 Agent，Server 到 Agent 之间的协议是开放且统一的，因此 Item 的名称或者 Key 是通用的。Zabbix 对于遵循 SNMP、JMX 协议等不依赖于 Agent 的监控对象，提供了尽可能开放的输入方式，可以满足我们 90% 的场景。

在某些特殊场景下，Zabbix 会出现问题，例如 JMX 的 Client 端的同名 Java 类在不同的版本下内容不同，这就必须采用新建 classloader 来实现，Zabbix 没有考虑到这一点，如需扩展还要进行二次开发。

#### 4. 用户界面与数据存储

Nagios 从系统管理员的经验出发，其信息记录的载体为一般的文本文件，信息的组织方式很清晰。信息的录入与维护是通过 VIM、Emacs 等编辑器直接创建与更新的。一套合理而规范的信息通过文本记录与通用编辑器维护符合系统管理员的习惯，它能够让我们在最短的时间内真正认识到程序数据结构的本质。文本化清晰明了的协议可产生好的实践，这是 UNIX 编程艺术哲学的观点之一。

Zabbix 的所有信息入口是面向一般用户而非系统管理员的，信息通过传统的 Web 用户界面录入，而非 Emacs 编辑器。虽然界面的用户体验有待提升，但它可让一个非系统管理员专业人士快速上手。其信息可以存储在 MySQL、Oracle 等主流数据库上，按照关系型方式存储数据为未来数据的组合查询提供高效而通用的工具。Zabbix 的信息对外除提供一般的 Web 界面操作外，还维护了一套开源 API，面向外部应用程序调用，在用户界面下了大功夫。

Nagios 依赖于强大的开源社区，为了解决信息存储与检索的问题，一个 Nagios 的附属项目主要用来将 Nagios 的配置信息和 Event 产生的数据存入数据库（目前支持 MySQL 和 PostgreSQL），以方便地实现对数据的快速检索和处理，并为通过 Web 接口程序来管理这些数据提供了保障。

NDOUtils 主要由以下 4 个部分组成。

(1) NDOMOD Event Broker Module (NDOMOD.O)：用来输出 Nagios 进程产生的数据（data 和 logic），其前提是 Nagios 在编译时开启了 Event Broker 功能。同时，NDOMOD 模块还可以导出与 Nagios 配置有关的信息（包括 Nagios 监控进程运行时环境产生的动态数据）



至文件、UNIX 域套接字或者 TCP 套接字。NDO2DB 将通过前面这三种方式获得 Nagios 的有关数据。

(2) NDO2DB: 用来接收由 NDOMOD 和 LOG2NDO 组件输出的信息并将之存储在数据库中。在启动时, NDO2DB 进程将创建一个 TCP 套接字或 UNIX 域套接字以监听客户端(输出端)的连接请求。目前仅支持 MySQL 数据库。

多个客户端可以同时向一个 NDO2DB 守护进程输出数据, 此时的 NDO2DB 将为每个连接进来的客户端(Nagios 实例)建立一个连接进程, 以实现每个客户端数据的独立存储、检索和处理。

(3) LOG2NDO: 用来将 Nagios 的历史日志通过 NDO2DB 进程输出至数据库。LOG2NDO 与 NDO2DB 进程通信的方法依然是标准文件、UNIX 域套接字或者 TCP 套接字。

(4) FILE2SOCK: 从标准文件或标准输入读入数据, 并将之输出至 UNIX 域套接字或 TCP 套接字。当 NDOMOD 或 LOG2NDO 将数据输出至标准文件时, 此工具可用来将这些标准文件中的数据读出并发送给 NDO2DB 进程监听的 TCP 套接字或 UNIX 域套接字。

NDOUtils 已集成到了 OMD 项目中, 将其烦琐的安装部署过程全自动化。如图 14-16 所示。

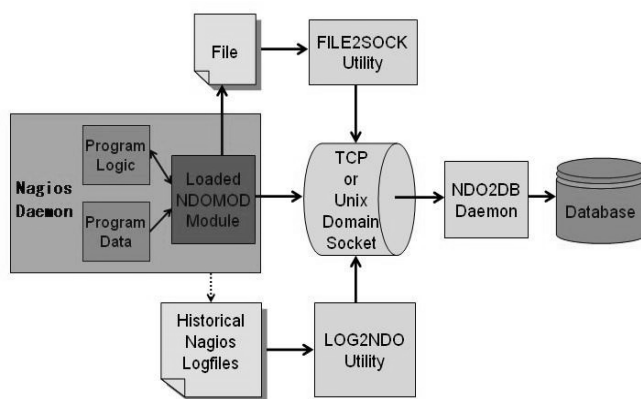


图 14-16 NDOUtils 组件图

Zabbix 一开始提供的就是面向企业级监控的平台, 很好地解决了一般用户与系统管理员的分工, 提供了良好的用户界面与信息存储形式。Nagios 之所以受到追捧, 更多的是其面向 UNIX 类系统管理员的衍生过程, 通过 Core 及其独有的面向接口编程、事件通知的 NEB, 吸引了大量的第三方项目支持, 逐步形成面向企业级的监控平台。OMD 与 Nagios Core 的并行存在反映的并不是一个可靠而易部署的工具, 恰恰相反的是开源之后围绕版权走向的分歧。Nagios 的商业目的是很清晰的, 其提供一个开放而简陋的调度程序作为 Core, 集成社区第三方工具形成一个完整的开源版, 但其要保证它足够的“玩具”性质。相反的是社区内希望在 Nagios 上发展出一个足以与商业产品抗衡的版本, 但由于版权限制和

开发周期不得不另外开辟站点，从而形成了 OMD。

是 Zabbix 还是 Nagios？我们先来假设一个前提，你的监控项是足够多的，你有一个比较大的数据中心，服务器的数量超过 500 台，监控项超过 1 万。在这个假设下，如果你是拥有天赋的系统管理员，并且提供专职监控岗位，被给予了足够的信任，则请选择 Nagios。在信任的基础上给予有能力的系统管理员足够的时间是可以换取一个灵活而适用的监控系统的，随着他和它的契合及标准的梳理，逐渐会形成一份你的企业独有的监控管理与工具集合。

什么？系统管理员一般，运维监控管理经验也一般，没有时间精打细磨。好的，如果希望快速出成绩，满足项目与领导的要求，立即抛出一份企业级的解决方案，那你肯定应该选择 Zabbix。简单的安装、部署方式，可用的用户操作界面，完备的 Agent 集合，足以在一个月之内满足你数据中心的要求，后续没有问题吗？后续的最大问题在于 Zabbix 太需要完善与一致性，如果你需要对它进行专业地客户化，与其他运维管理系统对接，以及对未来的性能瓶颈时，那么你将需要一个独立运维团队对其维护。另一条路如同使用微软 Windows 操作系统一样，免费体验良好的标准版，之后再放手购买 Zabbix 的服务来解决客户化需求，将运维的乐趣完全交给 Zabbix 的服务提供商。

### 14.2.4 商业与开源：最后的决策

商用的运维工具能够一蹴而就地满足我们的期望吗？商业产品能够在短时间内满足我们的暂时性需求，商业监控软件的购买不仅仅是购入了一套产品，还能够帮助我们领悟到一套监控管理的方法论，但随后而来的是什么呢？随着你的监控内容、任务越来越明晰，对细节控制的要求会越来越高，商业软件此时的弊端也就尽显无疑。对于那些品牌公司的商业软件来说，我们还是在不考虑动辄几百万的产品、服务费用的基础上考虑问题。当你在招标采购时，所有销售人员会告诉你他们的产品支持所有监控内容，他们的监控系统是最强大的，一旦选型完毕，产品即可入住，你可能为了增加一个定制化的 Oracle 脚本监控而被要求支付几十万的客户化费用，这又是另一条规则，IT 人员的价值远远高于硬件、软件，当然是优秀 IT 人员，或者是一名普通的 IT 人员站在了封闭的软件系统之上。商用软件的最大弊端是成本昂贵与定制化的不及时。前者可以通过淘汰品牌产品来规避，而后者则更多地依赖于监控产品公司的技术实力：在产品的设计时是否考虑到了足够的灵活性，是否采用公有协议而兼容社区插件，是否有一个强大的售后服务团队。

较之大品牌企业商业的监控软件及解决方案，国内著名厂商提供的工具有以下几方面优势。

（1）中西文化差异造成工具使用习惯上的很大差别，西方文化强调标准流程，企图通过统一固化的流程来消除人为导致的不确定因素，习惯于照章办事，宁愿损失效率，也要严格执行规定的流程步骤。而中国文化强调工具为人服务，或者说流程服务于人，习惯于

用巧劲办事，四两拨千斤，重视效率。

(2) 要求通过工具简化流程。

(3) 简化工作，重视工具的界面设计，要求通过工具的设计减轻人们在工具操作上花费的时间。

(4) 国内厂商开发的工具关注于国人文化之所需，在用户体验及流程裁剪上更适宜。

(5) 有过 ERP 实施经验的人都会将购买的国外知名品牌软件经过一轮客户化的项目实施，最终变成新的产品，而且不好用，这也是文化差异导致的关注度差异。

在服务支持上，除非是国际知名大品牌公司，一般技艺精湛但不专攻于国内市场的公司其服务支持水平和及时性都无法得到保证，通过电话、邮件或者远程桌面支持解决问题都是非常低效的。而国内厂商最怕的是贴牌供应商，有着品牌光环的公司外包一家技术创新的小公司，前者负责营销，后者专攻产品，在涉及服务支持时，二者在关系很的混杂，并不会专注于售后服务的建立，在选择商用产品时要避免以上两种情况。

## 14.3 JMX监控原理解析

JMX (Java Management Extensions, Java 管理扩展) 是一个为应用程序、设备、系统等植入管理功能的框架。JMX 可以跨越一系列异构操作系统平台、系统体系结构和网络传输协议，灵活地开发无缝集成的系统、网络和服务管理应用 (百度百科)。通俗地讲 JMX 为 Java 进程提供了一个可从外部远程管理 JVM (Java Virtual Machine) 内的对象的接口。管理动作包括查看、修改对象属性。JVM 进程在启动后将需要管理的资源封装起来，注册到一个 JMX 服务管理中心。远程的管理者遵循 JMX 标准来管理这批资源。

Java 领域的创始人及先驱们有着不一般的野心。Java 语言本身的解释型性质，借助于 JVM 成为 “一次编写，到处运行” 的跨平台语言。随着网络时代的兴起，Applet 依托浏览器实现了最厚重的 Web 2.0，在网页上实现一切桌面应用效果。JMX 同样被人们抱着统一的野心被设计出来，其并不是一门所谓的规范标准，其最终目的是对一切资源进行规范管理，包括硬件、网络、操作系统及其上的进程。

我们先来对 JMX 中定义的标准术语进行介绍，之后预览 JMX 体系接口。在后面的具体实现及详细学习中还会不断回顾这些内容，在这里让我们统一进行理解。

### 1. 受管资源 (Mangeable resource)

受管资源是一切可以被 Java 访问或封装的应用、设备或实体。在本书里它们就是我们监控的一切对象。这些对象通过 API、命令行甚至其他协议暴露给 JMX 来管理。受管对象可能是一台网络设备，也可以是一个数据库。

## 2. MBean

MBean (Managed Bean) 是 JMX 概念中的核心，它是一个普通的 Java 类，遵循了 JMX 规范，按照一定的要求定义属性与方法。无论受管资源是什么，最终都是通过 Java 语言进行包裹并放入到 MBean 中的，通过访问 MBeans 的属性、调用其方法来实现管理目的。MBeans 的类型有三种：标准 (Standard)、动态 (Dynamic)、模型 (Model)，每一种 MBeans 都有自己的特征，在本书中我们只讨论标准 MBean。

## 3. MBean Server

MBean 实例化对象之后，要向 MBean Server 进行注册，这样才能被外部对象找到。MBean Server 也是一个 Java 类，作为一个注册中心，它负责管理一组 MBean 对象，并不直接将 MBean 对象引用递送给外部访问者，而是通过规范接口对外提供服务。另外，它还向其他对象提供对 MBean 的事件监听和注册。

## 4. JMX Agent

JMX Agent 是一个独立的 Java 进程，是一个完整的容器，在里面创建 MBean、包裹受管资源、注册 MBean Server，完整的资源集合在容器内产生。除此之外，JMX agent 还负责定义与配置合适的“暴露”方式，可以通过 SNMP、HTTP 方式或纯粹的 Java RMI 方式对外暴露受管资源。

## 5. Protocol Adapters and Connectors

我们如何与外界发生关系？通过定义好的协议封装我们的数据，我们可以将 JMX 的受管资源数据转换到 HTTP 上，也可以转换到 SNMP 上。适配器 (Adapter) 负责将一个信息接口变换成客户端所期待的一种接口，从而使原本因接口不匹配而无法在一起工作的两个对象能够在一起工作。Adapters 负责在不同的协议上传送数据，而连接器 (Connectors) 为特定的连接方式工作，例如可以启动一个 RMI 连接器，监听特定的端口，等待 RMI 客户端的请求。

### 14.3.1 JMX的体系结构

JMX 的体系结构可分为四个层次。

(1) 设备层 (Instrumentation Level)：主要定义了信息模型。在 JMX 中各种管理对象以管理构件的形式存在，在需要管理时向 MBean 服务器注册。该层还定义了通知机制及一些辅助元数据类。

设备层是我们要管理的一切对象吗？如何用 Java 对一切对象进行管理呢？设备层是最接近监控对象的一层，它是一层适配器 (Adapt)，它在 Java 与监控对象之间进行转换，例如你若要监控数据库，则监控方式会通过检查特定的数据表并查看其状态来发现。设备层首先通过 JDBC 直接连接数据库获取我们的数据，之后将状态映射到 MBean 中，符合 JMX

规范。类似于数据库，操作系统、硬件等都可以按照这种方式，通过本地方法与 Java 的转换变成一个服务规范的 MBean。

在这里提一个问题，如果数据库可以直接通过 SQL 查询数据表监控，网络设备通过 SNMP 监控，则我们还有必要转换成 JMX 对象来监控对象吗？答案当然是没有必要。

(2) 代理层 (Agent Level): 主要定义了各种服务，以及通信模型。该层的核心是一个容器，包装了所有的核心内容，所有管理构件都在其中完成注册，这样才能被管理。在 MBean Server 上管理构件，并不直接和远程应用程序进行通信，通过协议适配器和连接器进行通信。而协议适配器和连接器也以定义与配置的形式在容器中发挥作用。代理层起到的作用是以一个容器的方式存在，集中所有已经过适配转换的受管资源对外提供服务。

(3) 分布服务层 (Distributed Service Level): 主要定义了能对代理层进行操作的管理接口和构件，这样管理者就可以操作代理。然而，当前的 JMX 规范并没有给出这一层的具体规范。这就意味着我们可以通过 HTTP、RMI 等任意方式访问、管理这些资源。

(4) 附加管理协议 API: 主要用来支持当前已经存在的网络管理协议，例如 SNMP、TMN、CIM/WBEM 等。

从 JMX 的体系结构 (如图 14-17 所示) 来看，它是一种可管理一切资源的架构与框架，但具体实现需要我们按照它的规约来填充。那么我们为什么还要使用 JMX 呢？Java 控制不了别人，但在内部，无论是 JVM 还是各大 Java 应用服务器，都提供了 JMX 的管理接口，也就是说它把需要管理的 MBean 对象及低层资源全部实现并打包好了，我们只需要掌握分布式服务接口就可以对它们进行管理和监控工作，包括一个 Java 普通进程、WebLogic、Tomcat 等。

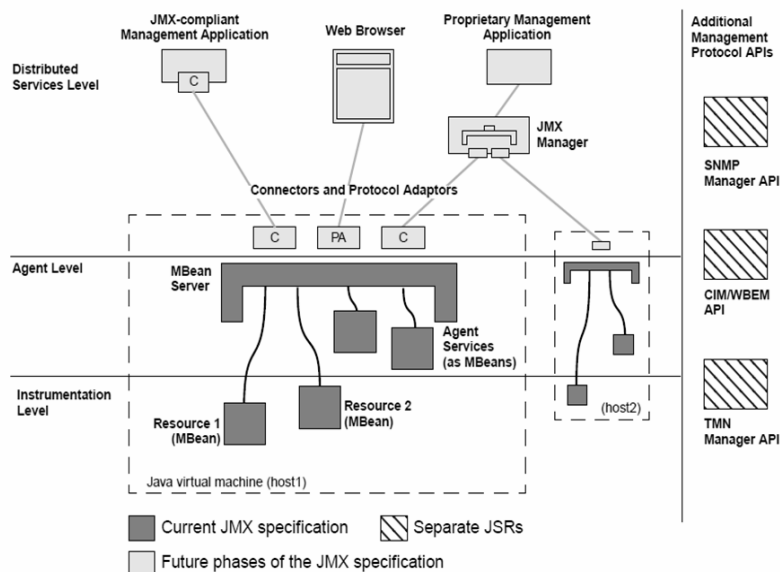


图 14-17 JMX 体系结构

### 14.3.2 一个完整的JMX体系架构实例

接下来让我们完成一个具体的 JMX 监控实现，这涉及体系架构中的每一个部件。假设你在 Windows 上使用过 Eclipse 编写最简单的 Java 代码，可以用 main 函数下打印出 “hello world”，有了这个 “基础”，我们可以完成两个 JMX 示例程序。

#### 1. 环境准备

我们首先上 Oracle 网站下载与 JMX 相关的包，它们分别是 JMX RI (1.2) 与 JMX Remote API (1.0.1)，下载的地址为 <http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-java-plat-419418.html>，下载完毕后我们将会在 zip 压缩文件中找到三个 jar 包，分别是 jmxtool.jar、jmxremote.jar、jmxri.jar。

在 JMX RI 包中提供了关于 Agent、设备层所要使用的接口类，其中包括一个 HTML adapter，专门用来给初学者示范使用。

JMX Remote API 提供了一些标准的远程访问方法。

#### 2. 管理你的资源

在这个例子中我们虚构一个受管理对象，它可以是一个数据库、网络或者打印机，这个对象通过一个叫作 MonitorResource 的类进行访问，而这个类继承了一个 MBean 接口类，对外提供了标准的访问方法。先来看看这个 MBean。

MonitorResourceMBean 清单如下。

```
package com.jmx;

public interface MonitorResourceMBean {

    public String getMemStatus();
    public String getCpuStatus();
    public String getBizStatus();

    public void setBizStatus( String bizStatus );
}
```

它定义了三个获取资源状态的方法，都是以 “get” 开头的，分别是获取内存、CPU，以及业务功能的状态。随后定义了一个 “set” 方法，它从外部设置业务功能的状态。实现这个接口的类被实例化从对象后将被注册到 MBean Server 中。

随后我们创建具体的实现类，这个类会和真正的监控资源打交道。

MonitorResource 清单如下。

```
package com.jmx;

public class MonitorResource implements
```

```

        MonitorResourceMBean {

    public String getMemStatus() {
        /*
         * do something for get memory status
         */
        return "memory status is ok ";
    }

    public String getCpuStatus() {
        /*
         * do something for get cpu status
         */
        return "cpu status is ok ";
    }

    public String getBizStatus() {
        /*
         * do something for get business status
         */
        return "biz status is " + bizStatus;
    }

    public void setBizStatus(String bizStatus) {
        this.bizStatus = bizStatus;
    }

    private String bizStatus = "ok ";
}

```

这个类的方法会保持与接口一致。值得注意的是大部分方法都直接返回一个字符串，但在实际环境中，其应当是将 JVM 或应用的自身状态返回。

最后我们创建一个 Agent，在这个容器中将把 MBean 注册到 Server 中，并创建一个 HTMLAdapter 对外提供访问接口。

MonitorResourceAgent 清单如下。

```

package com.jmx;

import javax.management.*;
import com.sun.jdmk.comm.HtmlAdaptorServer;

public class MonitorResourceAgent {
    private MBeanServer mbs = null;

    public MonitorResourceAgent() {

```

```
mbs = MBeanServerFactory.createMBeanServer( " Monitor
ResourceAgent " );
HtmlAdaptorServer adapter = new HtmlAdaptorServer();
MonitorResource mr = new MonitorResource();
ObjectName adapterName = null;
ObjectName monitorResourceName = null;

try {
    adapterName = new ObjectName(
        " MonitorResourceAgent:name=htmladapter,port=4848 " );
    mbs.registerMBean(adapter, adapterName);
    adapter.setPort(4848);
    adapter.start();
    monitorResourceName = new ObjectName(
        " MonitorResourceAgent:name=monitorResource1 " );
    mbs.registerMBean(mr, monitorResourceName);
} catch (Exception e) {
    e.printStackTrace();
}

}

publicstaticvoid main(String args[]) {
    MonitorResourceAgent agent = new MonitorResourceAgent();
}
}
```

这个类创建了 MBean Server、Adapter MBean，以及我们自定义的 ResourceMBean，继承打包后对外提供服务。

编写完毕后我们直接执行 MonitorResourceAgent，并在本地浏览器中输入 `http://localhost:4848`，访问如图 14-18 所示所示页面。



图 14-18 Mbean 演示界面 1



在 Agent View 页面中共注册的 MBean 有三个，我们选择所关注的 MonitorResource，进入 MBean View，如图 14-19 所示。

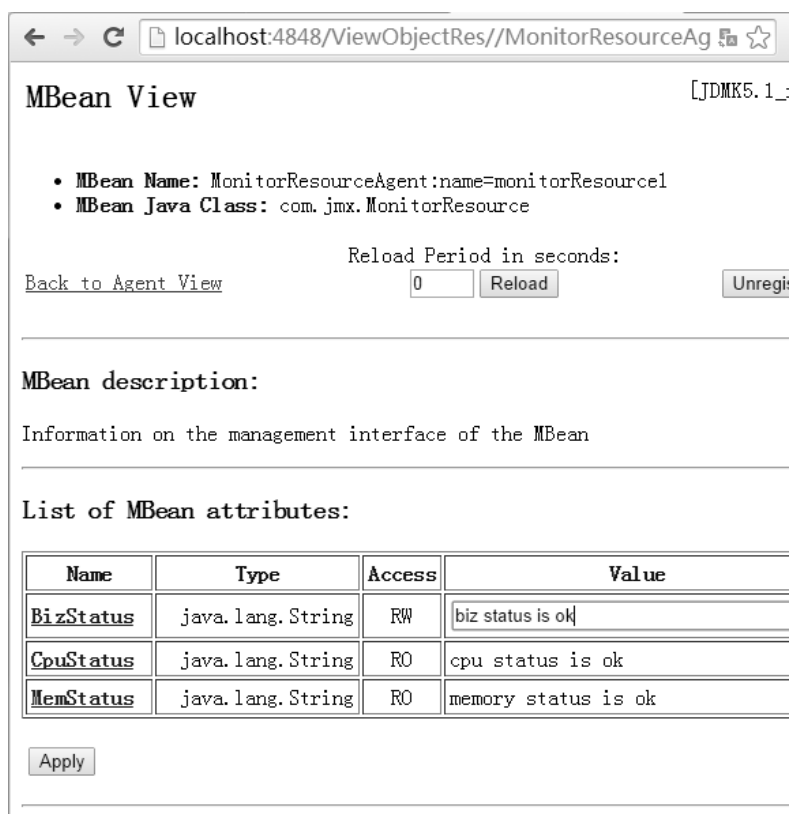


图 14-19 Mbean 演示界面 2

在页面上显示了我们所定义三个状态，我们在其中的 BizStatus 文本框中输入相应的值“very good”，之后单击“apply”，页面显示如图 14-20 所示。

<u>BizStatus</u>	java.lang.String	RW	biz status is very good
------------------	------------------	----	-------------------------

图 14-20 Mbean 演示界面 3

通过上面的例子，我们看到可以通过 JMX 体系结构从外部去观察，以及改变 JVM 中对象的属性，从而达到管理对象的目的。

### 14.3.3 通过 JMX 访问 WebLogic Server MBean

上一节通过一个例子看到 JVM 内部是如何将受管资源暴露并提供远程访问的，在这个例子中使用了 HTML 的 Adapter，采用 HTTP 对外提供服务。在本节我们关注的是如何作为

一个客户端去访问 WebLogic Server 这个中间件应用服务器，由于 WebLogic 提供了自己的连接器，遵循 T3 协议，所以在连接时要配置其自身的客户端 JAR 包。

### 1. 为远程客户端设置连接器类

为了在我们的远程监控程序上访问 JMX，请在该监控组件的类路径中包含下列 JAR 文件：WL\_HOME\lib\wljmxclient.jar

其中 WL\_HOME 是 WebLogic Server 的安装目录。

此 JAR 文件包含 Oracle 对 HTTP 和 IIOP 及其 WebLogic 专用 T3 协议的实现，它是一个连接器客户端。通过该实现，JMX 客户端随其连接请求发送登录凭据，而 WebLogic Server 安全框架对这些客户端进行身份验证。只有通过身份验证的客户端才能访问在 WebLogic Server MBean 服务器中注册的 MBean。

### 2. 建立到 MBean 服务器的远程连接

每个 WebLogic Server 域包括三种类型的 MBean Server，其中每一种 Server 提供对不同 MBean 层次结构的访问。

要连接到 WebLogic MBean 服务器，请执行下列操作。

1) 通过构造 javax.management.remote.JMXServiceURL 对象描述 MBean 服务器的地址。将下列参数值传递给构造方法。

(1) 作为与 MBean 服务器通信所用协议的下列值之一：t3、t3s、http、https、iiop、iiops。

(2) 承载 MBean 服务器的 WebLogic Server 实例的监听地址。

(3) WebLogic Server 实例的监听端口。

(4) MBean 服务器的绝对 JNDI 名称。JNDI 名称必须以 /jndi/ 开头，且后跟表 4-1 中所述的 JNDI 名称之一。

2) 构造 javax.management.remote.JMXConnector 对象。此对象包含 JMX 客户端用来连接到 MBean 服务器的方法。JMXConnector 的构造方法如下：javax.management.remote.JMXConnectorFactory.connector(JMXServiceURL serviceURL, Map<String,?> environment)

将下列参数值传递给构造方法。

(1) 在上一步中创建的 JMXServiceURL 对象。

(2) 包含下列名值对的散列映射：

```
javax.naming.Context.SECURITY_PRINCIPAL, admin-user-name
javax.naming.Context.SECURITY_CREDENTIALS, admin-user-password
javax.management.remote.JMXConnectorFactory.PROTOCOL_PROVIDER_PACKAGES,
"Weblogic.management.remote"
```

Weblogic.management.remote 包定义可以用于连接到 WebLogic MBean 服务器的协议。远程 JMX 客户端必须在其类路径上包括此包中的类。

(3) 通过调用 `JMXConnector.getMBeanServerConnection()` 方法连接到 WebLogic MBean 服务器。该方法返回 `javax.management.MBeanServerConnection` 类型的对象。MBeanServerConnection 对象是到 WebLogic MBean 服务器的连接。可以将它用于本地连接和远程连接。

(4) Oracle 建议在客户端完成其工作时，通过调用 `JMXConnector.close()` 方法关闭到 MBean 服务器的连接。

### 3. 一个示例程序

下面的示例程序用于连接到一个 WebLogic 的 Domain Runtime MBean Server，并使用 `DomainRuntimeServiceMBean` 为域中的每个 `ServerRuntimeMBean` 获取对象名。然后，它检索并输出每个服务器的 `ServerRuntimeMBean` 的名称和状态属性值。

除了 `connection` 和 `connector` 全局变量，类将 WebLogic Server 服务 MBean 的对象名分配给一个全局变量。类内的方法将频繁地使用这个对象名，它在定义后不需要更改。

`PrintServerRuntimes()` 方法获取 `DomainRuntimeServiceMBean` `ServerRuntimes` 属性的值，该属性包含域中所有 `ServerRuntimeMBean` 实例的数组。

`PrintServerState` 代码清单如下。

```
package com.jmx;

import java.io.IOException;
import java.net.MalformedURLException;
import java.util.Hashtable;
import javax.management.MBeanServerConnection;
import javax.management.MalformedObjectNameException;
import javax.management.ObjectName;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;
import javax.naming.Context;
public class PrintServerState {
    private static MBeanServerConnection connection;
    private static JMXConnector connector;
    private static final ObjectName service;
    // 实例化 DomainRuntimeServiceMBean 对象名
    // 这样可以通过类使用此对象名。
    static {
        try {
            service = new ObjectName(
```

```

        "com.bea:Name=DomainRuntimeService,Type=Weblogic.management.
        mbeanservers.domainruntime.DomainRuntimeServiceMBean");
    } catch (MalformedObjectNameException e) {
    throw new AssertionError(e.getMessage());
    }
}
/*
 * 实例化与 Domain Runtime MBean Server 的连接
 */
public static void initConnection(String hostname, String portString,
    String username, String password) throws IOException,
    MalformedURLException {
    String protocol = "t3";
    Integer portInteger = Integer.valueOf(portString);
    int port = portInteger.intValue();
    String jndiroot = "/jndi/";
    String mserver = "
Weblogic.management.mbeanservers.domainruntime";
    JMXServiceURL serviceURL = new JMXServiceURL(protocol, hostname,
        port, jndiroot + mserver);
    Hashtable h = new Hashtable();
    h.put(Context.SECURITY_PRINCIPAL, username);
    h.put(Context.SECURITY_CREDENTIALS, password);
    h.put(JMXConnectorFactory.PROTOCOL_PROVIDER_PACKAGES,
        "Weblogic.management.remote");
    connector = JMXConnectorFactory.connect(serviceURL, h);
    connection = connector.getMBeanServerConnection();
}
/*
 * 打印一组 ServerRuntimeMBeans。
 * 此 MBean 是运行时 MBean 层次的根。
 * 此域中的每个服务器承载自己的实例。
 */
public static ObjectName[] getServerRuntimes() throws Exception {
    return (ObjectName[]) connection.getAttribute(service,
        "ServerRuntimes");
}
/*
 * 迭代 ServerRuntimeMBean，获取名称和状态
 */
public void printNameAndState() throws Exception {
    ObjectName[] serverRT = getServerRuntimes();
    System.out.println("got server runtimes");
    int length = (int) serverRT.length;
    for (int i = 0; i < length; i++) {
        String name = (String) connection.getAttribute(serverRT[i],
            "Name");
    }
}

```

```

        String state = (String) connection.getAttribute(serverRT[i],
"State");
        System.out.println("Server name: " + name + ".Server
state: "
        + state);
    }
}

public static void main(String[] args) throws Exception {
    String hostname = args[0];
    String portString = args[1];
    String username = args[2];
    String password = args[3];
    PrintServerState s = new PrintServerState();
    initConnection(hostname, portString, username, password);
    s.printNameAndState();
    connector.close();
}
}

```

## 14.4 SNMP监控原理解析

### 14.4.1 SNMP协议解析：MIB库与消息类型

SNMP（Simple Network Management Protocol）即简单网络管理协议，它为网络管理系统提供了底层网络管理的框架。SNMP 协议的应用范围非常广泛，在诸多种类的网络设备、软件和系统中都有所采用。

相对于其他种类的网络管理体系或管理协议而言，SNMP 在当时更易于实现。SNMP 的管理协议、MIB 及其他相关的体系框架能够在各种不同类型的设备上运行，包括低档的个人电脑、高档的大型主机、服务器，以及路由器、交换器等网络设备。

一个 SNMP 管理代理组件在运行时不需要很大的内存空间，因此不需要太强的计算能力。SNMP 协议一般可以在目标系统中快速开发出来，所以它很容易在面市的新产品或升级的老产品中出现。尽管 SNMP 协议缺少其他网络管理协议的某些优点，但它设计简单、扩展灵活、易于使用，这些特点大大弥补了 SNMP 协议应用中的其他不足。

SNMP 采用了 Client/Server 模型的特殊形式：代理/管理站模型。对网络的管理与维护是通过管理工作站与 SNMP 代理之间的交互工作完成的。每个 SNMP 从代理负责回答 SNMP 管理工作站（主代理）关于 MIB 定义信息的各种查询。

SNMP 代理和管理站通过 SNMP 协议中的标准消息进行通信，每个消息都是一个单独的数据报。SNMP 使用 UDP（用户数据报协议）作为第四层协议（传输协议）进行无连

接操作。SNMP 消息报文包含两个部分：SNMP 报头和协议数据单元 PDU。如图 14-21 所示。

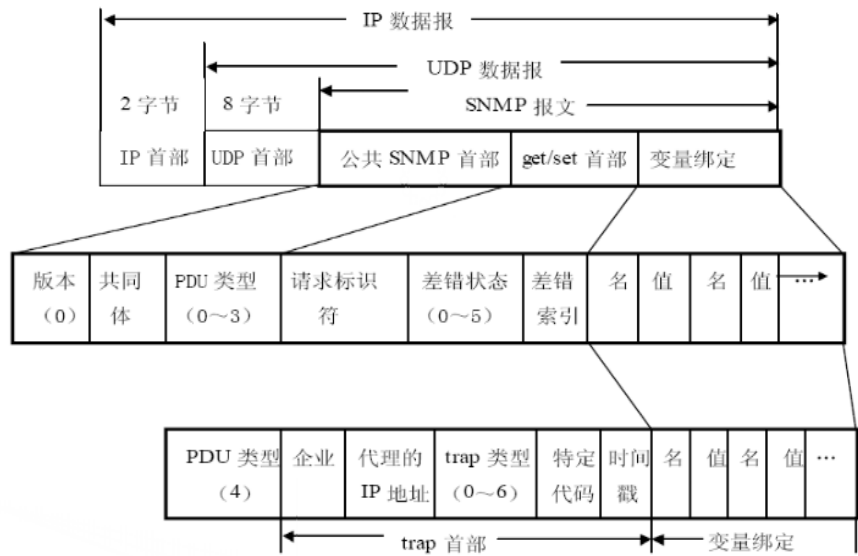


图 14-21 SNMP 数据报结构

- **版本 (Version Identifier):** 确保 SNMP 代理使用相同的协议，每个 SNMP 代理都直接抛弃与自己协议的版本不同的数据报。
- **共同体 (Community Name):** 用于 SNMP 从代理对 SNMP 管理站进行认证；如果将网络配置成要求验证，则 SNMP 从代理将对团体名和管理站的 IP 地址进行认证，如果失败，则 SNMP 从代理将向管理站发送一个认证失败的 Trap 消息。
- **协议数据单元 (PDU):** 其中 PDU 指明了 SNMP 的消息类型及其相关参数。

1. 管理信息库 MIB

IETF（互联网工程任务组）规定的管理信息库 MIB 定义了可访问的网络设备及其属性，其由对象识别符（OID: Object Identifier）唯一指定。MIB 是一个树形结构，SNMP 协议消息通过遍历 MIB 树形目录中的节点来访问网络中的设备。

管理信息库 MIB 指明了网络元素所维持的变量（即能够被管理进程查询和设置的信息）。MIB 给出了一个网络中所有可能的被管理对象的集合的数据结构。SNMP 的管理信息库采用和域名系统 DNS 相似的树型结构，它的根在最上面，没有名字。

对象命名树的顶级对象有三个，即 ISO、ITU-T 和这两个组织的联合体。在 ISO 的下面有 4 个结点，其中一个（标号 3）是被标识的组织。在其下面有一个美国国防部（Department of Defense）的子树（标号是 6），再下面就是 Internet（标号是 1）。在只讨论 Internet 中的对象时，可只画出 Internet 以下的子树（图中带阴影的虚线方框），并在 Internet

结点旁边标注上{1.3.6.1}即可。在 Internet 结点下的第二个结点是 mgmt（管理），标号是 2。再下面是管理信息库，原先的结点名是 mib。1991 年定义了新的版本 MIB-II，故结点名现改为 mib-2，其标识为{1.3.6.1.2.1}或{Internet(1) .2.1}。这种标识为对象标识符。如图 14-22 所示。

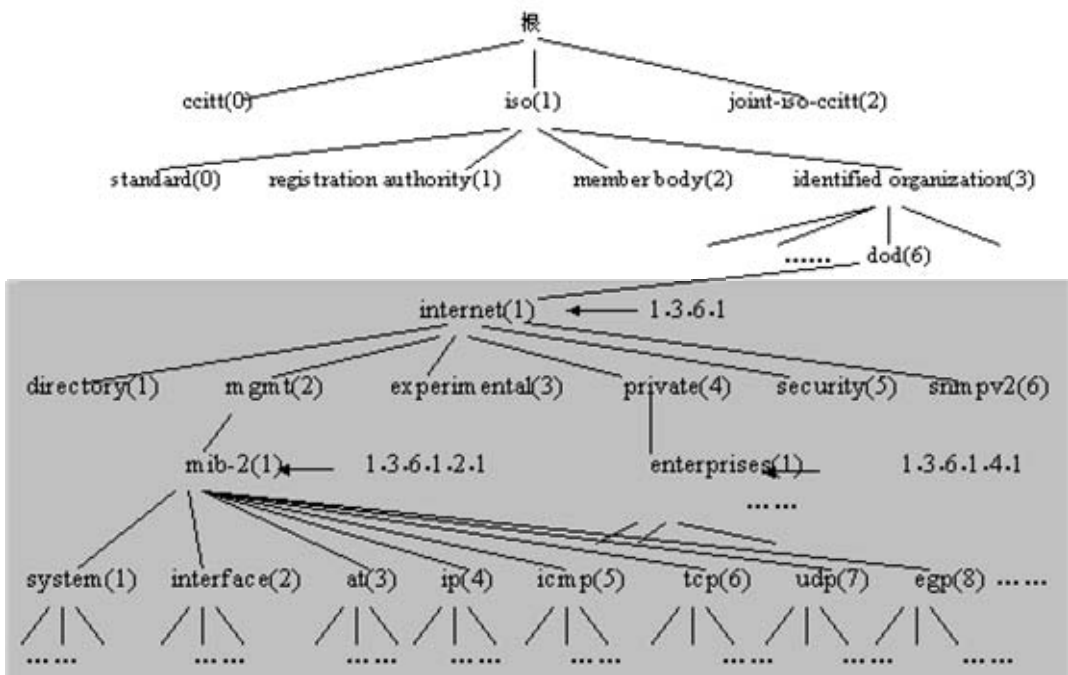


图 14-22 管理信息库

最初的结点 mib 将其所管理的信息分为 8 种类别，如表 14-1 所示。现在 de mib-2 所包含的信息类别已超过 40 种。

表 14-1 结点 MIB 管理的信息类别

类 别	标 号	所包含的信息
system	(1)	主机或路由的操作系统
interfaces	(2)	各种网络接口及它们的测定通信量
address translation	(3)	地址转换（例如 ARP 映射）
ip	(4)	Internet 软件（IP 分组统计）
icmp	(5)	ICMP 软件（已收到 ICMP 消息的统计）
tcp	(6)	TCP 软件（算法、参数和统计）
udp	(7)	UDP 软件（UDP 通信量统计）
egp	(8)	EGP 软件（外部网关协议通信量统计）

应当指出，MIB 的定义与具体的网络管理协议无关，这对于厂商和用户都有利。厂商可以在产品（如路由器）中包含 SNMP 代理软件，并保证在定义新的 MIB 项目后该软件仍

遵守标准。用户可以使用同一网络管理客户软件来管理具有不同版本的 MIB 的多个路由器。当然，一个没有新的 MIB 项目的路由器不能提供这些项目的信息。

## 2. SNMP 的五种消息类型

在 SNMP 中定义了五种消息类型：Get-Request、Get-Response、Get-Next-Request、Set-Request 和 Trap。

### (1) Get-Request、Get-Next-Request 与 Get-Response

SNMP 管理站用 Get-Request 消息从拥有 SNMP 代理的网络设备中检索信息，而 SNMP 代理则用 Get-Response 消息响应。Get-Next-Request 用于和 Get-Request 组合起来查询特定的表对象中的列元素。例如，首先通过下面的原语获得所要查询的设备的接口数：

```
{iso org(3) dod(6) internet(1) mgmt(2) mib(1) interfaces(2) ifNumber(2)}
```

再通过下面的原语进行查询（其中第一次用 Get-Request，其后用 Get-Next-Request）：

```
{iso org(3) dod(6) internet(1) mgmt(2) mib(1) interfaces(2) ifTable(2)}
```

### (2) Set-Request

SNMP 管理站用 Set-Request 可以对网络设备进行远程配置（包括设备名、设备属性、删除设备、使某一个设备属性有效或无效等）。

### (3) Trap

SNMP 代理使用 Trap 向 SNMP 管理站发送非请求消息，一般用于描述某事件的发生。

## 14.4.2 使用SNMP4J实现服务器监控

上面讲述了大段的理论知识来说明 SNMP 到底是什么，SNMP 是一种用来管理网络设备的协议，其信息按照树形结构保存，而交互这些信息的类型有五种。

本节通过做一个示例来帮助你直观地认识 SNMP，我们并不需要从网络连接层开始一行一行地实现代码。互联网上有很多开源的共享库供我们使用，以让我们关注于 SNMP 本身，当然，有了前面高性能服务部分的内容，你当然可以实现一个无阻塞 I/O 版本。在这里我们直接 SNMP4J，它是一个用 Java 来实现 SNMP 协议的开源项目。其项目地址为 <http://snmp4j.org/>。

环境准备的第一步是监控机器，对，就是你家或公司的办公 Windows 电脑，所有的重要设备都会提供 SNMP 服务。

笔者使用的是 Windows 7 操作系统，单击“控制面板”，选择“程序和功能”，在左连栏单击“打开或关闭 Windows 功能”，在弹出的菜单中将“简单网络管理协议（SNMP）”勾选上。如图 14-23 所示。



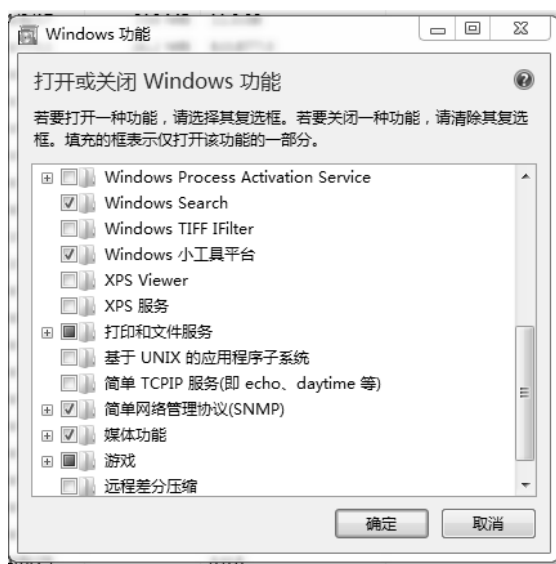


图 14-23 Windows 功能窗口

在“服务”面板中，检查我们的 SNMP Service 是否已经启动。如图 14-24 所示。



图 14-24 SNMP Service 面板

双击该服务，进入具体的服务管理面板，选择“安全”标签，添加一个“public”社区，并且选择“接受来自任何主机的 SNMP 数据包”。如图 14-25 所示。

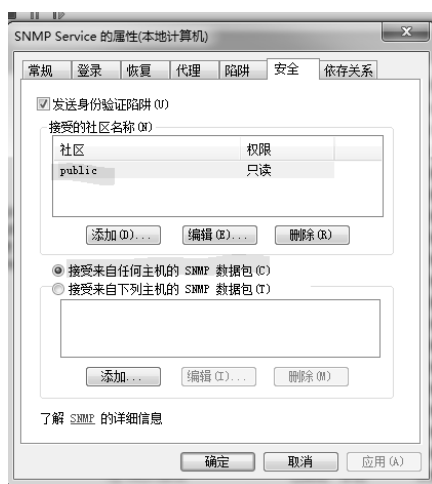


图 14-25 配置接收 SNMP 数据包

到 <http://www.snmp4j.org/html/download.html> 上下载 SNMP4J 包，将 snmp4j-2.3.1.jar 添加到你的 Eclipse 项目工程中。

```
package com.snmp;

import java.io.IOException;

import org.snmp4j.CommunityTarget;
import org.snmp4j.PDU;
import org.snmp4j.Snmp;
import org.snmp4j.Target;
import org.snmp4j.TransportMapping;
import org.snmp4j.event.ResponseEvent;
import org.snmp4j.mp.SnmpConstants;
import org.snmp4j.smi.Address;
import org.snmp4j.smi.GenericAddress;
import org.snmp4j.smi.OID;
import org.snmp4j.smi.OctetString;
import org.snmp4j.smi.VariableBinding;
import org.snmp4j.transport.DefaultUdpTransportMapping;

public class SNMPCClient {

    Snmp snmp = null;
    String address = null;

    public SNMPCClient(String add) {
        address = add;
    }

    public static void main(String[] args) throws IOException {
```

```

    /**
     * 在构造函数中输入我们要监控的对象地址，这里我们监控的是本机
     */
    SNMPClient client = new SNMPClient( "udp:localhost/161" );
    client.start();

    /**
     * OID - .1.3.6.1.2.1.1.1.0 => SysDec
     * OID - .1.3.6.1.2.1.1.5.0 => SysName
     * 抓取系统描述字段信息
     */
    String sysDescr = client.getAsString(new OID( ".1.3.6.1.2.1.
1.1.0" ));
    System.out.println(sysDescr);
}

/**
 * 打开一个 snmp session
 */
private void start() throws IOException {
    TransportMapping transport = new DefaultUdpTransportMapping();
    snmp = new Snmp(transport);
    transport.listen();
}

public String getAsString(OID oid) throws IOException {
    ResponseEvent event = get(new OID[] { oid });
    return event.getResponse().get(0).getVariable().toString();
}

public ResponseEvent get(OID oids[]) throws IOException {
    PDU pdu = new PDU();
    for (OID oid : oids) {
        pdu.add(new VariableBinding(oid));
    }
    pdu.setType(PDU.GET);
    ResponseEvent event = snmp.send(pdu, getTarget(), null);
    if (event != null) {
        return event;
    }
    throw new RuntimeException( "GET timed out" );
}

private Target getTarget() {
    Address targetAddress = GenericAddress.parse(address);
    CommunityTarget target = new CommunityTarget();
    target.setCommunity(new OctetString( "public" ));
    target.setAddress(targetAddress);
}

```

```
        target.setRetries(2);
        target.setTimeout(1500);
        target.setVersion(SnmpConstants.version2c);
        return target;
    }
}
```

控制台的输出是：Hardware: AMD64 Family 21 Model 2 Stepping 0 AT/AT COMPATIBLE  
- Software: Windows Version 6.1 (Build 7601 Multiprocessor Free)。

### 14.4.3 Linux下的监控实现：NET-SNMP

在 Linux 下我们常使用 NET-SNMP 进行类似的操作，它是一个专用于 SNMP 的软件集，包括客户端的库文件、命令行集合、可扩展的 SNMP Agent、Perl、Python 模块。

在 NET-SNMP 下提供的命令行工具主要有以下几种。

**Snmptranslate：**在数字与文本之间翻译 MIB oid 的名称。

**Snmpget：**通过 SNMP GET request 与网络设备进行交互。

**Snmpgetnext：**通过 SNMP GETNEXT request 与网络设备进行交互。

**Snmpbulkget：**通过 SNMP GETNEXT request 与网络设备进行交互。

**Snmpwalk：**通过 SNMP GETNEXT request 找到管理节点下的一颗子树。

**Snmpbulkwalk：**通过 SNMP GETBULK request 找到管理节点下的一颗子树。

**Snmpset：**通过 SNMP SET 与网络设备进行交互。

**Snmptrap：**发送一条 SNMP TRAP 或 INFORM 通知消息。

**Snmptrapd：**为一个 SNMP 后台 Daemon，负责监听发送过来的 SNMP TRAP 或 INFORM 通知消息。

下面是在 Linux 下执行 snmpwalk 获取一台域名为 myserver 的 Linux 服务器的系统信息。

```
$ snmpwalk -t 5 -Os -c demopublic -v 2c myserver system
sysDescr.0 = STRING: Linux myserver 2.6.18-164.el5 #1 SMP Thu Sep 3
03:28:30 EDT 2009 x86_64
sysObjectID.0 = OID: netSnmpAgentOIDs.10
sysUpTimeInstance = Timeticks: (14198303) 3 days, 11:39:23.03
sysContact.0 = STRING: Net-SNMP Coders
sysName.0 = STRING: test.net-snmp.org
sysLocation.0 = STRING: Undisclosed
```

### 14.4.4 MIB库浏览工具：ManageEngine

你可能会问，我们就看一个设备的 MIB 就这么困难吗？仅仅为了系统描述要写那么多代码？要不就是敲击繁琐的命令？代码和命令的功能是让 SNMP 的监控集成到我们的平台中，如果纯粹是为了通过 SNMP 查看设备，则有更加方便的开源工具可以使用。

SNMP Mib Browser 是卓豪 ManageEngine 推出的一款免费的 MIB 浏览器工具，用于监控采用 SNMP 协议的网络设备和服务器。利用 MIB 浏览器，你可以加载查看设备的 MIB，执行 GET、GETNEXT 和 SET SNMP 操作。这款免费工具的使用方法非常简单，允许你查看、配置和解析 SNMP Trap。

下载并安装这套工具，在左边栏选择你需要 get 的 mib oid 号，在右侧输入监控对象信息，包括 host、port 及 community，通过快捷键“Ctrl+G”同样可获取作者本机的描述信息。如图 14-26 所示。

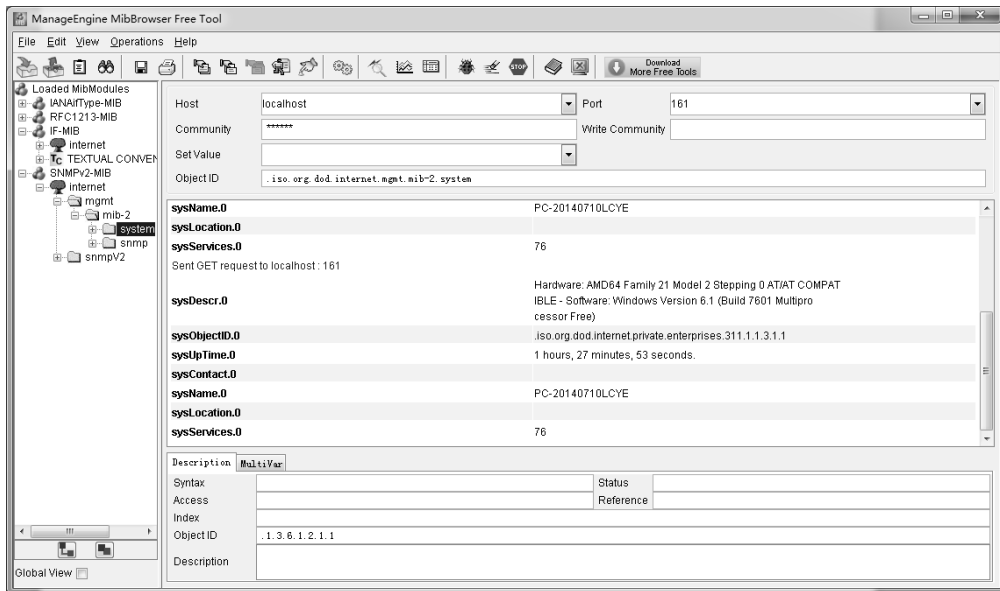


图 14-26 ManageEngine 的 SNMP Mib Browser

# 运维管理

前面的内容让我们的注意力全部集中在美妙而生动的创造与建设之中，通过将最原始的指令、数据聚合，形成现实世界中的业务的支撑体。如同在 IT 技术的海洋中潜游，自由地寻找漂亮的珊瑚、贝壳，静心观赏鱼群的活动，一切都是如此宁静与美好。很可惜，此刻我们必须浮出海面，除了纯粹的技术实现，海平面上的其他东西一样值得我们关注，可能你会抱怨这个世界的干扰因素太多而无法专心做好一件事，我们可以在这里停下来，但有几个原因值得我们继续往下走：

（1）所有技术都是为业务服务的，专注于技术并没有过错，你在享受过程，优雅的语言、精妙的逻辑及最终实现会一次又一次打动你，当然在享受之余拥有一份不错的薪水和福利就再好不过了。如果此时你不再享受，略有疲惫，则请回到本章，它会尽可能地帮你找到答案，并认识到将 IT 运维作为一门手艺与一份工作的区别。

（2）科学点亮了人类文明之灯，IT 技术建立在基础科学之上。但我们是科学家吗？不是，或者不纯粹是，但可以肯定的是我们将科学带来的文明传递给现实世界，将其运用到生活之中。我们应当投入部分精力看看科学的传递是如何被有效组织的，关注终端用户之所需，以及这种需要又是如何被满足的。

（3）如果你感冒严重去医院治疗，则你肯定不会关注什么是细菌与病毒，它是如何侵入到人体细胞并破坏免疫力的，相反，你可能关注挂什么科，在哪儿打针开药，什么时候病情可痊愈，有时还需要护士的人性关怀。IT 运维工程师可以说与外科医生做的是一件事，但在现实环境下我们必须关注除此以外的事儿，即这些手艺工作者们的运转方式。

## 15.1 服务级别管理，IT 与业务的一致性

工程师们聚集在一起，在不同岗的位上用自己的专业技能支撑起整个 IDC 的运营，我们看到的是服务器、网络设备、操作系统，以及运行在其上的中间件。你可能花了大部分时间来参透 IDC 各类组件的运行原理，随着组织的需要，你的技能不断提高，但你所获得

的认可是否也同步增加了？很显然，答案没那么简单，没有谁会愿意为你的个人技能的提高而买单，除非给组织带来了直接收益，并且收益能够被量化，被人所感知。很明显大部分运维工程师从来没有关注过如何量化自己的产出，让客户满意自己的服务，这已超出了技术的范畴。

小春是一个德才兼备的 IT 运维工程师，工作勤奋努力，经常会加班加点地解决客户的问题，他处在一个疲于奔命的状态中，哪里有需要就到哪，收到一个电话或一个邮件就会立即响应，如同消防员，但尽管如此，其还是时不时地受到客户的指责。

小春为企业内部的分部职场做接入层交换机配置，他提前准备好相应的配置管理信息，使用基本命令快速地将 10 台交换机统一配置好，并通过了网络连通性调试，整个过程耗费 5 个小时。小春在当时的情况下已经尽了自己的最大努力了，但还是没有得到分部职场人员的认可，客户说：“分部职场有很多人员都在等着使用电脑，耗费 5 个小时，会浪费我们多少宝贵的时间呀！”。小春彬彬有礼地回答：“下次我一定想办法加快速度”。

回去后小春认真思考，琢磨着通过技术手段来解决这个耗时问题，他用了 3 天写了一个本地输出+com 接口的终端程序，能够自动连接到交换机进行标准化配置，这样的话处理时间由之前的 5 小时缩短到了 30 分钟。

很快另一个分职场也在做网络组建，功夫不负有心人，小春同学利索地拿起了他的新工具，在 30 分钟内以迅雷不及掩耳之势配置完了所有交换机，分职场的业务人员非常满意：“不错，其实很简单嘛，以后就按照这个标准做”，随后陆续有这类网络配置的需求找上门来，但偏偏有一次对方的网络设备全部是国产品牌，在命令集及规约上和原来的程序不匹配，这一折腾 3 个小时过去了。小春再一次受到了客户的批评。

“为什么处理这么慢啊，上次按照我们的要求改进后已经可以在 30 分钟内完成了呀？”

“这次的网络设备在兼容性上出现问题，因此多花了一些调试时间。”

“你知道这个调试使我们多长时间无法工作吗？你好好反省下，我要投诉。”

等待小春的依旧是客户的不满意，小春立即崩溃。小春该怎么办？哀求客户原谅他的过错？或者向客户诉说他的苦衷，请求谅解？客户表示无法理解，于是与客户据理力争，说明运维服务已经做得足够好了。最后问题直接升级，说客户要求苛刻、无理取闹，从而引发了一个更大的危机。

### 15.1.1 客户满意度与期望

上面的例子是个人与客户之间的纠纷，类似的情况实际上往往在部门、公司之间发生。在现实的 IT 运维管理中，类似的情况实在太多，无论运维系统如何努力，都无法完全获得客户的认可，让客户满意。

“IT 与业务融合”是 IT 治理追求的目标之一，但 IT 与业务似乎很难找到一个让双方能够顺畅沟通的方式，IT 与业务之间有太多“说不清、道不明、剪不断、理还乱”的关系，要解决“IT 与业务融合”的难题，必须找到一个在所有情况下都适用的最佳实践，在所有情况下都说得清、道得明、清晰明了，最终让我们的客户满意。反过来，我们要找到使客户不满意的深层次原因，以及让客户满意的金钥匙。

客户满意度由客户实际体验与客户期望对比而产生，我们用下面的公式对客户满意度进行定义：

客户满意度 = 客户体验 - 客户期望

如果我们不知道客户的期望，则无论我们怎么做都无法确保客户满意。我们可以不断提升客户体验，但这要综合考虑成本因素，当客户体验提升到一定程度无法再提升时，我们有没有办法对客户期望进行管理？

客户及所有人都只关注自己当前所从事的事情，而他们对 IT 的期望是：

- 核心应用系统稳定，7×24 小时可访问；
- 应用数据永不丢失，数据备份自动完成；
- 应用业务功能使用顺畅，上手即用，无须使用指引手册；
- 应用系统响应速度快，无须任何等待；
- 应用系统开发、部署、安装交付速度快，提出需求后马上满足；
- 办公网络通常，可以访问任何角落的资源；
- 办公电脑的使用不出现延迟；
- 邮箱、磁盘空间无限大，可以发送任意大小的文件，并且又快又好；
- 7×24 小时随叫随到的 IT 服务支持，现场解决一切问题。

客户并不关注操作系统的高深哲学、网络的源远流长、服务器的分发模型，只会关注能不能让用户用、什么时候可以用、不可用或不好用会影响我们多少业务量。

IT 运维人员的期望是：

- 请求、事件、变更量适度，拥有思考时间；
- 客户对自己技术能力、水平的认可；
- 客户对问题解决的过程充满感激；
- 具备挑战性任务，在工作中得到成长；
- 跟上时代潮流，用新技术解决常规问题；
- 研究源代码、系统架构、部署体系，不断沉淀。



我们无法将客户的期望与运维工程师们进行匹配，并不是客户不关注 IT，而是 IT 本身的出发点是为了更好地支撑业务，再好的技术技能也不一定能够获得客户的认可，使客户满意。

客户期望最初是建立在自己的想法之上的，不经过有效的沟通、交流与培训，客户很难理解 IT 所提供的服务。IT 需要从另一方面帮助客户理解 IT，将期望与真实服务的距离缩短，这就是运维工程师们很少看到的成文约定，在 IT 与业务上形成理解的一致性，就必须在提供任何实质性服务之前对服务质量进行协商，之后签订成文协议，双方达成一致意见。

服务级别管理（Service Level Management）是对 IT 服务的供应进行谈判、定义、评价、管理，以及以可接受的成本改进 IT 服务质量的流程。所有这些活动都要求在业务需求和技术快速变化的环境中进行。服务级别管理试图在服务质量的供应与需求、客户关系和 IT 服务成本之间找到某个合适的平衡点。

对于 IT 运维人员来说，认识到运维属于服务的范畴，而服务是一个供应和接收的过程是很重要的。在所有服务的交互之前，通过服务级别管理将真实服务水平以可度量的标准发布出来，管理好客户预期。

服务级别管理确保客户需要的 IT 服务得到了持续的维护和改进。这主要是通过针对 IT 部门的运作绩效进行协商、监控和报告，以及在 IT 部门及其客户之间建立有效的业务关系来实现的。

有效的服务级别管理可以改进客户方业务运作的绩效并因此产生更高的客户满意度。由于 IT 人员更加清楚他们被期望提供什么和他们可以提供什么，因此他们可以更好地计划、预算和管理他们所提供的服务。

服务级别管理的意图在于确保 IT 组织中所有的运营服务和绩效都可以按照一致的专业的方式来测量，同时服务和创建的报告都能够满足业务和客户的需求。其目标是确保按照约定的级别来提供 IT 服务，并且将来的服务也是按照约定的标准进行交付的，也将通过主动测量来发现和实施对所交付服务级别的改进。

服务级别管理由三部分内容组成：服务目录、服务级别协议（SLA）与操作级别协议。如图 15-1 所示。

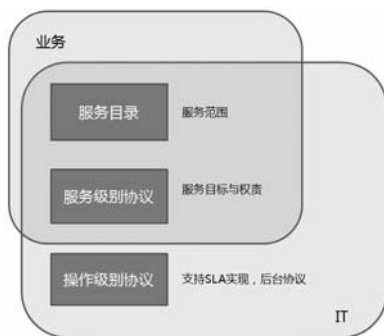


图 15-1 服务级别管理的三个组成部分

## 15.1.2 服务目录——IT服务的菜谱

服务目录是对 IT 运维组织中正在使用的所有服务的记录，该记录应当包括：服务（用业务语言表达）、交付服务过程中所使用的 IT 服务组件、企业用户的服务优先级，以及服务用户数量等。服务目录中的信息应该是易于管理、切实可行的。服务目录决定了 SLA 约束的服务范围，即服务商到底提供哪些服务，只有客户“选择”了服务目录，IT 才可以进行报价与服务，服务目录之外的内容不受 SLA 限制。

服务目录简单来讲就是包含了所有服务的列表。服务目录的内容包括服务的名称、定义、状态（生命周期）、服务的承诺等信息。

服务目录分为以下两个方面。

- 业务服务目录：包含了所有交付给客户的 IT 服务和这些服务与业务单元，以及依赖于这些 IT 服务的业务流程之间的关系，是从客户视角理解的服务目录，这是服务目录的客户视图。
- 技术服务目录：包含了所有交付给客户的 IT 服务和支持服务、共享服务、组件和 CI 等将服务提供给业务所必需的支持。它应支持业务服务目录。这是从技术服务的视角理解服务目录，而且它并不是客户视图的一部分。通过技术服务目录可以帮助 IT 部门全面反映其自身的情况，通过业务服务目录可以将其自身呈现为一个 IT 服务提供者。如图 15-2 所示。

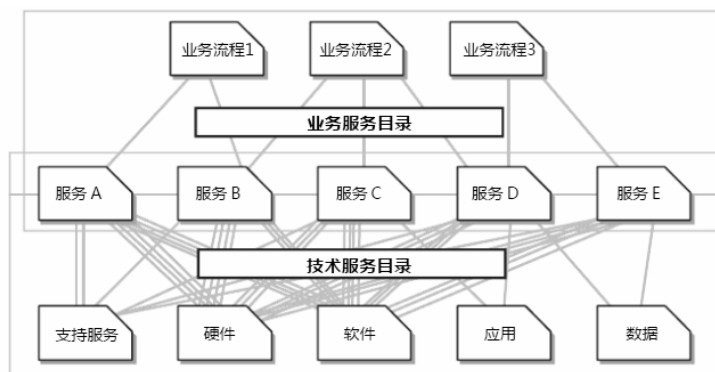


图 15-2 业务服务目录与技术服务目录

服务目录实际可以理解为一个菜馆里的菜谱，描述了客户想要的服务（菜）。其中可能包含服务的时间、服务的简介、如何能获得服务、服务的分类等内容，尽量包含客户所关心的内容。而客户所关心的主要是如何获得价值，这有两重含义：一个是功用 Utility（产品或服务为满足某些特定需要或者去除某些阻碍而提供的功能），一个是功效 Warranty（对产品或服务能满足约定要求的承诺或者保证）。在服务目录中，一般而言，功效是较难体现的，除非是非常标准化的服务，否则建议无须列得太细，这样会导致在编写 SLA 的过程中

服务提供方会陷于被动局面。而在服务目录中能获得较好展现的就是功用方面。通过服务目录中的服务，服务提供方能提高客户的某种能力或者加强客户的某种资源。

15.1.3 从宏观到可操作性的服务

1. 服务级别协议（Service Level Agreement）

服务级别协议是对服务目录中描述的内容的进一步说明，它是服务提供商与客户之间的一种书面协议，规定了服务需要达到的主要目标和双方具体的责任，是有效的衡量考核工具。

在建立服务目录后，必须设计最合理的服务级别协议构架，以确保覆盖所有的服务和 IT 系统的用户。构建 SLA 主要有以下两种方法。

1) 基于服务

制订的每一个服务级别协议针对一个服务，除非不同的用户对同一个服务有各自相同的特殊要求。在这种情况下，对同一个服务级别协议需要设立不同的指标体系。在签署服务级别协议时，需要考虑到用户的范围，让不同的用户范围代表签署。或者可以分开签署不同的协议来避免一些不必要的麻烦。如表 15-1 所示。

表 15-1 服务级别指标

系统名称	移动互联网上交易系统		
服务内容	系统简介	移动互联网上交易系统	
	系统关键交易	综合相关产品 移动终端投资	产品移动终端投资
服务级别指标	指标名称	目标值	
	服务支持时间	重大事件支持:7×15（周一至周日 8:00~23:00） 日常事件支持：5×8（周一至周五 8:30—18:00） 值班电话支持：7×24	
	维护时间	系统计划维护区间集中在：周一至周日 23:00~8:00 的区间内 每月计划维护时间小于 16 小时	
	可用性	目标值	>= 99.85%
	可靠性	按重大事件标准定义	
		目标值	<=5 次/半年
	可连续性	RPO	8 小时
		RTO	36 小时
	性能	目标	关键交易页面大小小于 70k 时，客户端平均响应时间小于 10s，缓慢操作百分比小于 20%。 页面大小（k）:客户端平均响应性能(s)=7：1，日均访问 500 次以上

续表

系统名称	移动互联网上交易系统		
	容量	车、财、意投保单数	最大月访问业务 150 万单，最大日提交数 15 万单
		访问量	单个 landingpage 页面 PV 不大于 10 万/天 单个关键页面 PV 不大于 5 万/天
	数据备份	xxx 数据库；RMAN	
	事件解决时间	遵照事件解决时间说明规定	
	服务请求解决时间	遵照服务请求解决时间说明	

## 2) 基于用户

确保一个服务级别协议只针对内部的单独的用户群后，这个协议将包括用户使用的所有服务，能够包含所有服务和所有用户。

从用户的角度来说，他们可能会倾向于这种协议，其所有需求都被包含在同一份文件里。一般只签一次字就可以了，这种比较简单，但是对服务级别管理项目推动小组来说，工作量可能会有所增加。

## 2. 操作级别协议（Operational Level Agreement）

操作级别协议用来支持 SLA 中的实现。OLA 是后台的协议，它定义的服务内容可能与客户不发生直接关系，却是实现 SLA 必不可少的。

OLA 在很大程度上和 SLA 很相似，但是定义的服务内容还是有较大差异的。在公司内部，人们关心的首要问题是公司的业务能不能正常运作。比如对于工作结算部门来说，他们关心的是确保公司的每个员工能够及时地领取其应得的工资，而不愿意过多地关心完成这一任务所必需的哪怕是关键的服务模块，比如网络设备、工资表单的处理软件或者打印机等。而 OLA 定义的就是这些用户不太关心的底层服务，比如网络是否连通、服务器硬件是否有问题等。

SLA 是 IT 与业务客户之间签订的一份 IT 服务保障协议，而 OLA 是 IT 内部的各个二级部门与 CIO 签订的一份用于量化保证服务质量而提供的承诺书，目的是为了保证 IT 实现 SLA 总目标。

可以说 OLA 是在 IT 内部的细化，为了支撑在 SLA 中承诺的服务，我们必须对组成服务的每一项基础设施进行级别定义，例如基础架构的需求处理时效、故障解决时效等都是作为指标下发到各 IT 部门的。

回到本章前面小春的例子，交换机的安装、调试、连通服务是定义在服务目录中的，对每一单的处理时长在提供该项服务之前就与业务方进行了协商与约定，如果将时间定义为 3 小时，并且客户认可此处理时效，那么在此时间内完成的工作都属于有效范围。

## 15.2 变更管理，使服务有效传递

服务级别管理为 IT 提供了面向业务的门面，而在 IT 内部有效地运作来实现服务级别所承诺的标准，则还需要有一套固定的流程、工具来保证。

用户会依据服务目录中的定义申请某项服务，而为了将服务递送给用户，在 IT 运维内部会引发资源的变化，我们将这种变化称为变更。

经验显示在 IT 运维中发生异常往往与变更有关系，造成这些异常的原因很多：人员疏忽、技能缺乏、未准确验证、没有影响度分析等。很多 IT 企业都忌讳非常规性变更，他们将其当作万恶之首、IT 运维的最大敌人。在互联网兴起、强调变化的时代，唯一不变的就是变，变更不能成为阻碍 IT 发展的障碍，我们要结合变更管理流程及工具，通过最佳实践保证变更完成得又快又好。

变更管理的目标是通过标准方法和过程保证用户服务响应快及高质量的。

### 15.2.1 变更控制的角色、阶段

#### 1. 角色与职责

在变更管理流程中有以下角色及相关职责：变更请求人、变更主管、变更审批组、变更实施人员。

##### 1) 变更申请人

原始需求人，将需求申请提交给变更主管。

##### 2) 变更主管

变更主管为具体变更的管理人员，对变更质量负责，对进度负责；

评估变更请求并规划实施，确保变更在预定的时间、资源和成本内完成；

主导变更方案和时间计划的制订，组织变更的规划、测试、实施和回顾；

在必要时确保回退计划得以正确实施。

##### 3) 变更审批组

根据变更影响范围的不同，可能由运营、基础架构、开发的各级管理人员组成；

审核提交的变更请求和支持性文件，并确保潜在的影响和风险得到评估，批准变更请求；

解决变更争议（当两个或两个以上变更发生时间冲突时）；

日常变更由变更经理在系统中审批。

#### 4) 变更实施人员

变更实施人员同时包括实施处理人和实施复核人；

根据变更主管制订的变更计划实施变更；

执行分派的任务以推进变更，总结变更实施情况并向变更主管反馈处理情况。

### 2. 变更管理的三个阶段

变更控制不能太简单，用户将需求递送过来后直接上生产环境的方式是不可取的。一艘火箭马上要升空了，科学家问：“最新的发送指令版本在哪里？”工程师从裤兜里掏出一张 3.5 寸磁盘：“刚刚写好，马上发布”。你能放心让火箭升空吗？

变更控制也不能太复杂，如果为了添加一个内网域名，要找上各个领域的专家分析、设计、评审、实施和测试，则 IT 运维的整体效率会大打折扣。另一方面，无论变更的复杂度、影响面如何，每一个都需要经过审核，会导致人员疲劳，从而出现疏忽，导致质量控制落空。

一个变更从受理到实施完毕需经过需求分析、变更设计、变更实施三阶段。

#### 1) 需求分析

变更主管受理用户的申请需求，与用户就需求内容进行沟通确认，此阶段的目的是尽最大可能消除歧义、修正需求内容。另一个功能是将需求打包处理，而不是分散到独立服务目录中，提升用户体验。

我们可以期待将服务目录设计得非常清晰、简单，在需求提交之前无须任何交互性沟通，直接进入审批环节，但在实践中发现，用户在选择服务目录及填写服务表单时均有困难，直接进入审批环节而不进行交互会导致多次返工。服务目录限制了一个需求的大小，而对于开放式的需求分析，变更主管可以就用户需求进行打包式处理，原来用户为了搭建一个环境可能要拆分成 5 个服务目录内容，走多次审批流程，而经过一次需求分析可将需求分拆任务在内部完成。

#### 2) 变更设计

在需求分析通过之后，变更主管开始就需求进行设计，将每个需求分解成许多执行步骤，每个步骤有足够的可操作信息如搭建环境，会标明每一条命令，以及执行此步骤的输入、输出。步骤关联到配置管理系统、知识库，便于变更实施人员在实施过程中查询。

#### 3) 变更实施

实施人员要严格按照变更实施计划及相关操作手册执行，如果要执行实施方案或操作手册之外的内容，则需要征得变更主管的同意。如果在实施过程出现异常，则按变更实施

计划中的相应应急或回退措施进行处理；如果在计划中无对应的措施则升级到变更主管。当变更异常或失败的严重程度超出变更方案预期或可控制范围时，应立即升级到变更经理获得决策。变更实施异常及其处理情况要求实施人员记录到变更管理系统中，方便变更回顾和在解决变更引起的问题时查阅。实施完成后，通知验证人员严格按照变更验证计划执行验证。

所有变更都经过了以上三个阶段，在具体操作的实践中，运维人员会非常不适应，在推广之初会遇到若干阻力。“等这些流程、步骤做完了，变更的具体工作早就实施完毕了”“为什么要做这些重复劳动”、“变更设计与实施是重复劳动，一个变更为什么要做两遍”。运维人员会发出他的真实心声。这是由于管理层与运维操作层的关注点不一致导致的，要解决二者之间的不认同，需要采取以下措施。

(1) 在工具上要做到足够易用，关注用户体验，所有重复的问题要在工具上解决，而不是人。

(2) 在需求分析与变更设计时进行分类，将标准的无风险的服务识别出来，走快速通道，直接执行。

(3) 对于快速通道之外的变更，一律要求坚持流程规范。

## 15.2.2 变更管理的六个原则

开发部门提了一个需求给运维分组，需求没经过审批、变更计划没有启动、没有问题升级与通知、白天非服务窗口时间就由运维人员直接实施了。天啊，太夸张了吧，这个过程有多少个漏洞啊？这样做变更怎么行？处理这个需求的依据是什么？

开发部门说急、非常急，但运维工程师并不了解需求，也未评估影响。需求紧急，应走升级通道。

开发部门同事反馈已在测试环境中做过了，百分百没有问题，这话可以相信吗？我们相信任何人，但在不同的环境下可能会有不同的影响，是否提前做好了验证与回滚方案呢？

自负的运维工程师认为什么都懂，但他们往往懂的仅是如何操作，却不明白会有什么影响，业务关联是什么，以及如何快速回滚。在运维领域有一种现象是技能越突出、效率越高的工程师，发生故障、异常的频率也越高。

无可厚非的是运维工程师的 80% 的精力是投入在技术上的，他们不可能记住所有的规范、流程及细节，特别是在各类特殊情况下的不同的处理方法，流程可以是完美的，但需要工具支持。对于变更管理的要求需要以大原则的方式倡导，做到简明扼要。

运维工程师在日常变更中的六大原则为：走变更、知明细、先配置、分批次、准验

证、快升级，运维工程师需要牢记这 18 个字，而不是复杂的流程规范。

### 1) 走变更

所有需求要全部进入变更流程管理系统中，而不能通过一封邮件、一个电话直接执行。

### 2) 知明细

清楚我们有哪些服务目录，用户的需求如何与目录匹配，是否有例外需求，变更模板中的每种技术实现的影响是什么，变更的风险级别是什么。

### 3) 先配置

在变更实施前保证配置先行。例如搭建一个新的网络区域，在实施具体的操作前，先将所有设备信息录入到 CMDB 中，设置好 CI 之间的属性与关联，避免配置遗漏、后续无据可查。

### 4) 分批次

对于变更对象多、涉及面广的变更，要将变更对象分批次、安排多次变更执行。

### 5) 准验证

实施完成后要有准确的验证方法，做好备份、做好回滚方案。

### 6) 快升级

中途出现异常，在一定时间内无法解决的，应尽快升级，让相关人等及时知情。

## 15.2.3 变更分类与风险定级

### 1. 常规与非常规

大部分服务是提前定义好的，这类服务引发的变更被称为常规变更，变更的实施可以遵循标准步骤和知识库的指引。而除这些常规变更外，也会因其他原因在定义的服务目录之外发起变更，例如 IT 内部为了降低成本、优化架构等也会对现有资源进行调整，例如网络拓扑的改变、带宽出口的限速等，我们称这类情况为非常规变更。

从风险的角度来看，常规变更的影响和风险较易控制，一般有如下特征：

- 有标准化的变更模板；
- 有知识库指引；
- 重复、日常化、量大；
- 影响和风险较易控制。



而非常规变更的影响范围广、风险较高，缺乏完善的变更模板。一般具有如下特征：

- 缺乏完善的变更模板；
- 需由资深经验的同事执行；
- 年度计划、一次性或若干次执行；
- 影响范围广、风险较高。

## 2. 优先级

优先级用来说明变更需要得到实施的优先顺序，按照其紧急程度（即变更需要采取多快的速度来执行）来安排优先级，如表 15-2 所示。

表 15-2 风险级别

序号		优先级	说明
1	紧急	变更需立即执行	
2	高	可以安排在最近一次变更窗口中执行的变更，需要尽快执行的变更	
3	低	对时间要求不高，或者可以放在再下一个变更窗口中执行	

## 3. 风险定级

变更风险等级依据变更影响范围的大小、是否中断服务、是否造成数据丢失、变更复杂度、验证和回退方案等方面评估，分为高、中、低三个技术风险等级。

对于低风险：

- 无须特别关注，无窗口限制，影响范围小；
- 无影响或无关联影响；
- 非生产变更；
- 或有 HA、单边影响；
- 或无数据丢失、服务中断；
- 或在维护窗口内。

对于中风险需严格审批，在窗口内实施。符合下述任何条件之一的为中风险：

- 紧急变更；
- 没有回退和应急方案；
- HA 同时实施，服务中断；
- 在服务窗口内；
- 技术不成熟；

- 没有测试过；
- 没有验证方案。

对于高风险，需汇报，沟通窗口，变更序列确认，需验证、监控。符合下述任何条件之一的为高风险：

- 架构变化；
- 新技术引入；
- 没有先例；
- 技能要求高、需厂商参与；
- 复杂度高；
- 影响重要系统（一类）；
- 变更涉及有容量瓶颈对象；
- 无法全面验证；
- 无法充分评估关联影响。

可以参考如表 15-3 所示的衡量因素来评估实施变更可能带来的风险，对于已知的变更类型，应事先按衡量因素定义好变更的技术风险等级。

表 15-3 变更风险衡量因数

评估维度	问 题	评 估 (是/否)	权重（1 风险很小，2 轻微风险，3 普通风险，4 较高风险，5 高风险）	加权评估结果
紧迫性	(1) 是否紧急变更			
复杂性	(1) 变更方案是否会改变现有架构			
	(2) 变更主管是否完全掌握变更所涉及的技术点			
	(3) 是否是成熟的变更方案，并有成功实施的同类变更案例			
	(4) 是否在测试或灾备环境中测试过			
	(5) 是否有回退和应急方案			
	(6) 回退方案是否能在 30 分钟内完成			
	(7) 验证方案是否完整有效			
	...			
影响度	(1) 变更是否会影响生产环境（评估的前提）			
	(2) 冗余的两个设备是否要在同一时间段做变更			
	(3) 在最坏的情况下,变更是否会导致数据损坏或丢失			

续表

评估维度	问 题	评 估 (是/否)	权重 (1 风险很小, 2 轻微风险, 3 普通风险, 4 较高风险, 5 高风险)	加权评估结果
	(4) 在最坏的情况下,变更是否会导致服务中断			
	(5) 在变更过程中出现故障的可能性程度? (1 低, 2 轻微, 3 普通, 4 较高, 5 高)			
	...			
其他决策参考因素	监管部门是否要求必须在指定时间内完成			
	该变更不实施是否会导致项目延时,产生严重影响			

对于特殊的变更类型，应参照衡量因素，在变更发起时定义风险级别，如表 15-4 所示。

表 15-4 风险估值

变更总体风险	估值区间
高风险	5,4
中风险	3,2
低风险	1

15.2.4 表单、步骤、模板与日历

成功的 IT 管理需要均衡考虑三大要素，即人员（People）、流程（Process）与工具（Technology），简称 PPT。而 IT 管理的实践往往重流程，忽视相应的工具，从而导致人员无法适应。仅仅考虑流程，将绝大部分时间、精力和资源投入流程上的畸形的应用模式往往产生畸形的效果，要么是流程无法落地应用，要么是运维人员没有精力做技术，期望流程可以解决一切问题。

前面谈到了很多变更控制流程、角色、原则，但要落实下来必须有强大的工具支持，在 IT 服务项目实施前就要把工具的适用性考虑进去。工具是给人用的，不是用来折磨人的，一定要注意均衡发展。

1. 交互表单

用户依据向导来找到自己所需服务的目录，在每个目录下有相应的表单帮助用户准确描述自己的需求。表单由各技术组自行定义，并且依据服务内容变化同步更新，保证足够的灵活性。表单内的字段分类有必填、可选与自动补全三类，表单与需求系统解耦，以配置文件的方式管理。如图 15-3、图 15-4 所示。

选择表单: Middleware

☐ Cyberark配置申请表 ☐ F5配置申请表 ☐ HAPProxy配置申请表 ☐ J2EEpatch申请表 ☒ J2EE应用首次部署申请表 ☐ J2EE资源申请及环境搭建表

☐ 整系统下线(包含DB用户下线) ☐ 整系统下线(仅下线应用环境) ☐ 整系统下线(包含下线库) ☐ 子系统下线 (不涉及DB及存储)

J2EE应用首次部署申请表

J2EE应用首次部署申请表

复制 删除

子系统名: 请选择... 提示: 请填写RSMS子系统清单的标准子系统ID

子系统ID: 提示: 请填写RSMS子系统清单的标准名称

维护联络人: 提示: 请填写域用户名

环境: ☐ 生产环境 ☐ 测试环境

部署环境名称:

ear或war包名:

部署方式:

特殊部署方式:

是否标准部署方式 ☐

配置UM ☐ DMZ区开放给外网用户访问 ☐

sso配置: 配置sso ☐

JDBC配置:

其他特殊配置:

说明:  
[1, 标准部署方式是一个ear包, jar包部署在appserver上面, war包部署在webserver上面]  
[2, CLASSPATH里面加载标准部署包, 若有特殊包请写入 {应用名}\_bowl\_classpath.env或者 {应用名}\_afwl\_classpath.env文件中引用]  
[3, UM标准配置各系列OID]  
[4, 其他特殊配置包括Startup&Shutdown Class, JMS, Virtual Host等配置]

预览

图 15-3 用户申请需求时需要填写对应的服务表单

表单管理

选择表单: J2EE资源申请及环境搭建表

修改名称: J2EE资源申请及环境搭建表 ☒ 激活

选择版本: ☐ ver.1 ☐ ver.2 ☒ ver.3

提交目标系统: CHANGE系统

产配公式: 1 提示: eg. 3 \* sum[A] \* count[B], A和B均为表单项的name属性, 大小写敏感, 顺序计算, 不支持括号优先。

表单类型: 所有人可见

变更关系模板: 请选择

需要创建变更: 是

表单权限: 多个角色以逗号分开, 如: 角色A, 角色B

审批链: 没有设定审批链!

表单内容: 编辑表单 测试表单

复制 删除

子系统名: 子系统清单列表 提示: 请填写RSMS子系统清单的标准子系统ID

子系统ID: 提示: 请填写RSMS子系统清单的标准名称

受益人:

环境类型: 生产环境 已有环境套数: 申请第3套及以上需提供规划支持部允许申请的EOA签

资源: 

网络区域	逻辑实体	平台类型:
		WebLogic10
		WebLogic10
		WebLogic10

NAS: (如有nas需求请另外添加nas表单进行信息录入) 

连接nas的应用逻辑实体	NAS逻辑实体	
		<input type="radio"/> 新建nas卷 (请填写Linux&nas
		<input type="radio"/> 使用已有nas卷 (已有nas
		<input type="radio"/> 新建nas卷 (请填写Linux&nas
		<input type="radio"/> 使用已有nas卷 (已有nas

图 15-4 每个服务目录表单由服务 owner 自行定义，并与需求系统解耦

一个需求要在变更主管与用户之间形成交互，对需求的内容进行确认与修正，最后定版进入审批环节。

2. 精确步骤

要保证每个变更实施的质量一致，就必须对变更步骤进行控制，保证所有的人的执行方式一致。如图 15-5 所示。

- 在进行变更设计时要定义好每个操作步骤，在步骤中描述清楚具体的实施方法。
- 步骤内容是动态的，在实施过程中实施人员可反馈、登记信息。
- 步骤之间可按照顺序、分支来执行。
- 步骤定义了明确的实施时间窗口，必须在实施时间窗口方可执行。
- 一个变更很可能跨越多个领域的专业人员，在步骤处理人中进行选择。
- 步骤与知识库是关联的，除了说明具体如何操作，还要讲述背后的实现原理。

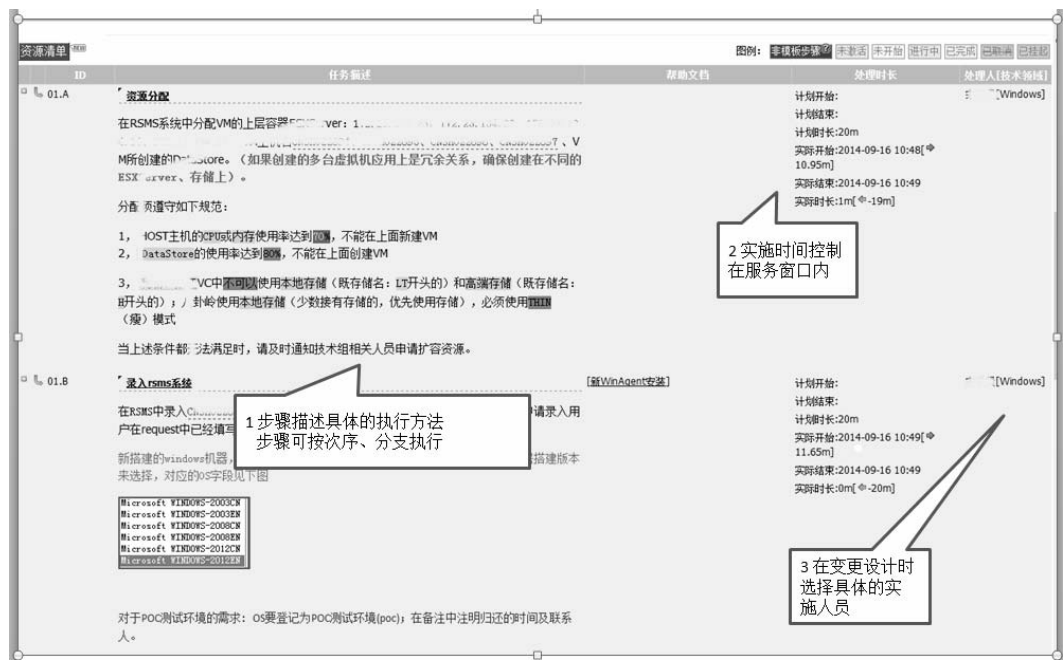


图 15-5 变更步骤

3. 定制模板

对于大多数已经标准化、但步骤复杂的变更，在工具上要考虑模板化，将一个变更序列固化下来，帮助运维人员快速启动一个变更计划。如图 15-6 所示。

[illegible]

图 15-6 定义一个变更模板，快速启动变更

#### 4. 变更日历

管理人员、运营人员甚至业务员常常会问当天要做哪些变更？变更有没有经过审批，有没有向上级汇报，有没有评估风险，有没有分析出关联应用？面对排山倒海的问题，运维人员常常要悉心整理这些问题，逐条耐心解答，将已知的问题复述多次，极大影响效率很低。变更系统内部需要有变更日历功能将这些信息聚合在一起。如图 15-7 所示。

**· 变更日期**

日期:

星期日	星期一	星期二	星期三	星期四	星期五	星期六
						1 2
2 2	3 28	4 42	5 59	6 67	7 43	8 8
9 1	10 65	11 54	12 43	13 19	14 5	15
16	17 1	18	19	20	21	22
23	24 1	25	26	27	28	29
30	31					

图 15-7 变更日历

每次的变更可能高达 60~70 个，如何从中找到管理层、需求方所关注的变更？这时需要在变更系统中设计一个人性化检索面板，通过“关联对象”“影响范围”把用户关注的变更全显示出来。如图 15-8 所示。



图 15-8 变更关联对象与影响范围

## 15.3 事件管理

事件管理即 IT 组织为客户解决其职责范围内的与 IT 组件相关的问题，恢复用户对组件的正常使用。事件管理的目标是快速恢复服务可用，消除对生产业务的影响，服务方们需要专业技能强、熟悉责任范围内的工作、沟通技能高的专家团队；客户方需要对问题表述准确，提供的信息要充分，在需要对问题进行检测时积极配合。

### 15.3.1 分类管理与评价体系

好的事件分类可以帮助用户找到解决问题的正确方法，找到正确的人进行处理。好的事件分类也可以帮助服务者对自己的工作进行有效分析及优化，通过技术、管理手段一次性清除问题。分类的通用方式按行政组织划分，各行政组织提供自己的服务目录。

#### 1. 事件前端分类

事件分类有适用于用户使用的服务目录分类，我们将这称为前端分类。在与用户签订的 SLA 中服务提供者可以详细描述所提供的事件服务，但事实上事件类目录很少在 SLA 中签订，因为我们很难预测到未来会有什么问题，SLA 只会从用户服务可用性的角度进行定义，事件处理是达成这个指标的一个有效手段。

我们在前端分类上有两种做法，各有优势。

- 精准命名：对每类事件设置一个精确命名，该类问题的处理方法由标准流程控制，

并且风险可控。在 ITIL 体系中将这类事件称为“服务请求”，其本质是用户对许可范围内标准资源的申请，既然是申请，则这类分类通道必然伴随着审批环节。精准命名的优势是能够帮助我们找到正确的人做正确的事，其缺点是无法涵盖所有的通道，即便涵盖，对于前端用户而言从大量的目录中选择自己的事件服务目录是很痛苦的。

- 宽泛命名：为了提升用户体验，服务提供者不会将名目繁杂的服务目录提供给用户选择，而是划分一个大行政组让用户将问题全部抛进来。平安运维在核心层面上使用了这类命名通道，甚至还会合并几个行政组建立横向条线，连贯地处理生产问题。例如：IT 部门分为网络、系统、中间件、DB 主机等，用户按照行政组上报事件，但一个核心应用的异常与一个用户终端的软件异常不一样，应用层面上的关联影响更多，中间件的异常可能与 OS 有关系，OS 的异常可能与网络有关系，为了应对这种场景，平安建立了跨技术组的横向团队，强调技能融合，一个人具备多种技术条线诊断技能，提高了问题处理的效率。

精准命名适用于问题明朗、服务标准、风险可控的服务请求，而异常不明确、需联合诊断的事件则采用宽泛命名。平安科技面向桌面终端用户的事件采用了精准命名，而面向应用核心服务的事件则采用了宽泛命名。

### 2. 事件后端分类

针对前端分类为粗略命名的情况，为了帮助服务提供者对问题进行分析从而找到优化方法，我们需要让事件处理人员在处理时或之后将问题放入正确的篮子中。后端分类的维度很多，例如按组件分类，但这种分类方式对运维优化的帮助并不太大，你很难从一个粗粒度数据中挖掘出有效信息。另外，可能因为某些项目、新组件、新架构可能引发大量的事件，在这种情况下分类就更困难了，我们推荐后端采用 tag 方式，自由打标签，通过开放的方式进行分类，标签是可检索、可补全、易创建的。从用户体验上看，为了避免处理人员重复输入，可以记录功能，在单击输入框后将最近 10 条类别显示出来；或者通过补全的方式，输入一个字符，将类似选项显示出来。

### 3. 评价体系

评价体系的目的是保证服务质量、注重客户体验，从而提升 IT 运维的竞争力。常用的手段有星级评价、客户回访等。服务对象是外部还是内部直接关系到评价体系是否实用，也关系到事件管理团队的侧重点是专业技能还是“外围”服务。为了提升客户体验，在事件处理的核心任务的外围总有一批“非核心”的工作投入，其内容包括：对问题进行分析、过滤、转交，ITIL 中的服务台起到了这个功用；回访、拜访、定期交流、商务沟通、合同界定是服务经理经常要安排的。另外专门会有一批质检人员对事件的处理时长、响应时长、用户评价、沟通记录等进行统计分析，并提出持续优化的措施。



如果公司面向的服务对象是外部用户，并且公司的收益直接取决于用户的满意度，那么“外围”服务会非常重要，我们需要在此领域部署一批专职人员。相反，如果面向的对象是内部人员，并且自身的团队成员规模不大，则把工作的重心放在对问题的解决之上，以最高的效率完成工作，在管理上尽量做轻做薄。团队强调的是个人技能，以培养专家为目的。

15.3.2 任务分发、协同与时效

1. 任务分发

表 15-5 显示了常用的三类任务分发模式。

表 15-5 任务分发模式

模 式	简 介	优 劣
统一队列	将事件放入到统一的队列中，由处理人员自行选择处理	劣势：无法保证事件处理的公平性，人员积极性难调动 优势：给员工一个轻松的问题处理环境
主动派发	将事件直接派发到当前手上事件量最少的在线人员身上，如果无在线人员，则将事件放入统一队列中。处理人员不能自行选择任务	劣势：工作量大时下，会给人员造成较大压力 优势：保证公平性，人员积极性易调动
银行排队	将事件放入统一队列中，在线处理人员将手中的事件处理完后，分发程序才将队列中的事件派发给处理人	优势：公平性只能得到部分保护，保留自行选单是为了提高工作效率，保证人员积极性。 劣势：无法彻底做到公平性

是否允许在线处理人员自行选择事件、是否自动派发到处理人员身上及派发条件决定了任务分发模式。平衡公平与效率是任务分发模式目标，如果 IT 系统做得足够灵活，则可以对模式进行动态调整，我们可以在不同的时机选择不同的模式以应对不同的需求。另外，权限功能会与任务分发模式紧密结合，哪些人可以受理哪些事件、哪些人可以自行选择等在系统设计时也要考虑得非常好。

2. 任务协同

处理人员在解决事件的过程中需要向其他人请求帮助，或者这个任务本身不是当前组的，需要转交到其他组。横向层面上的协同方式有两种：转交和协作。纵向层面上的协同方式为：升级，直接转入问题管理中。

3. 处理时效

在事件管理中需要对不同级别的事件进行处理时效的定义。

表 15-6 是一个简单的时效约定表。

表 15-6 事件处理时效

环 境	紧急程度	解决时间	解决时效要求
生产	生产在线等	1 天	85%在时效内解决
生产	生产普通	2 天	
测试	测试在线等	4 小时	
测试	测试普通	1 天	

15.3.3 内部上报要求

在 IT 内部，运维人员收到的与 IDC 核心服务相关的事件一般来自于开发、运营部门，这是典型的 DevOps 工作环境，其中的合作比传统环境中的合作更加紧密，一个问题的解决需要开发、运维团队的协作。为了提高内部交流的效率，双方约定一份上报规范对上报事件的处理有很大的帮助。

以下是一份约定宣导手册。

1) 确定自己无力解决问题

作为一名称职的开发、运营人员，你应为眼前的问题进行至少 30 分钟的思考，并确认自己无力解决。大多数问题是因为粗心导致的，比如多写了一个空格、配置文件关联错误等。有时依靠别人也解决不了问题，最终还要靠自己细心处理。

2) 搜索现存知识库

你遇到的问题很可能已经有相关的详细论述了，在外部你可求助于谷歌、百度，在内部你可使用事件系统、知识库的检索功能。绝大多数常见的技术问题都可以通过搜索得到满意的解决办法。

3) 认真准备问题

上面的两种办法都试过了，如果还是解决不了问题，那么你可以认真准备问题来上报事件了。在提问之前，你需要提供以下信息。

- 提供准确的配置信息

你必须提供准确的配置信息，仅给出子系统名或贴一段异常堆栈是不行的。如果你真的关心这个问题，那么你应当把子系统环境、所在主机、异常实例名、日志目录、关联关系、访问域名等配置信息全部整理清楚。

- 详细描述问题现象

作为事故现场第一人，也作为异常的最大“受害者”，你才能提供最有力的“证据”来帮助捉拿真凶。你应当停下来冷静地回想这个异常，以及与这个异常相关的过程、线索、信息，发挥你的文采，把问题描述清楚。

#### 4) 过程中全力配合

对问题的处理随时需要你的帮助，比如提供配置文件、临时借用账号、协调用户验证、提供源代码等，除了以上帮助，实际上最需要的是你的理解、尊重和信任。

#### 5) 事件上报中严禁的行为

- 三言两语、范围模糊

三言两语、范围模糊的提问是非常不礼貌、不负责任的行为。如果你无法准确、详细地描述问题，则不能让别人顺利地解答你的问题。

- 非友好方式

我们非常希望杜绝以下非友好方式的提问，而将精力集中在对问题的解决上。

“A 环境没问题，B 环境有问题，肯定是动了什么”“前面用得好好好的，为什么现在不能用了，肯定是动了什么”“你们是专家，所以只有靠你们了，你们必须解决”这些提问方式本身没有太多帮助，反而会给人造成条件反射性的抵触情绪，我们应该专注于事件本身。

- 重复上报事件

你可能会为了一个紧急问题而心急如焚，于是多次重复上报同一个 case，并分发到不同的处理人手中，希望加快处理进程。你要知道 case 资源是有限的，加快了你的处理速度后也会降低别人的处理速度，并带来不必要的重复劳动。

### 15.3.4 重大事件处理

每当重大事件发生时，需要将相关人员集中在一起，我们将这种联合诊断的过程赋予了 UIOC (urgent incident operation center)、War Room 等各种名称，其目的在于快速调动 IT 资源，高效协同诊断事件，在这个过程中，开发关注应用逻辑、运营关注业务影响、运维关注底层资源、DBA 关注数据库。

流程启动的第一步是将大家召集就位。沟通工具、渠道有多种，面对面沟通、邮件列表、即时通讯、视频会议等，不同团队类型有不同的处理习惯。但在事前我们就应当将这些通道提前建立，并验证随时可用。

UIOC 是一个联合诊断、积极配合过程，通常会有一个经验丰富的人员来现场指挥、协调各团队间的工作。

#### UIOC 六步骤

UIOC 流程启动后，如没有统一管理，则很容易陷入到一片混乱中，我们一般会参照下面六点按顺序进行问题分析：

### ● 问题描述

启动 UIOC 后，会对问题、异常进行一个简单描述，如 xx 系统的 xx 功能无法使用。另外高层会关注业务影响，在这个步骤中，运营人员应当迅速地抽取出业务变化率。

### ● 应用架构

在问题、业务影响描述清楚后，下一步是系统负责人对应用的整体部署架构进行说明（对于问题所在模块一目了然的这步可省略）。

这个整体部署架构中包括了主要的配置信息、关联方等，其主要目的在于缩小问题范围。

### ● 版本变更

依据应用架构的输出来判断在这个范围内是否有组件版本发布、基础资源变更。

大部分故障都是由“变”而起，不是外部（访问量、安全攻击），就是内部（版本、变更），该步骤帮助我们发现内部变化，如若找到相关影响对象，可以考虑准备回滚步骤、方案。

### ● 信息收集

以上三步应当是习惯性地快速完成，如仍无法准确定位到问题点，极有可能陷入到僵持状态中。信息收集阶段，各团队开始各自挖矿，开发人员查看用户访问量、应用异常日志，运维人员检查基础资源情况，包括性能数据、日志信息，DBA 检查数据库等待事件、top sql 等，再将各自发现的可疑点共享出来，尽可能形成问题关联，比如存储发现 IO 延时比较高，请 DBA 确认是否有影响（不是所有的延时都影响数据库）。

### ● 行为决策

步骤中确定的是快速恢复方案，UIOC 强调的是快速恢复，而不是问题分析，亦即找到问题点后快速采取恢复方案，而不是将时间耗费在穷根问底。UIOC 准确地说是发现问题点在哪里，而不是回答为什么会有这个问题点，对于已发现的问题点，应当问：是否可主备切换？是否可功能降级？是否可快速扩容？是否可版本回滚？

### ● 实施验证

在决策完毕后，实施方案，并做好验证，确保系统恢复正常。



图 15-9 UIOC 六步骤

## 15.4 人员管理：开放与分享

### 15.4.1 企业社交管理

很多技术人员在沟通表达能力上都有所欠缺，我们大部分时间都在与书本、机器交流，缺乏与人沟通的机会。笔者与很多 IT 服务供应商打过交道，供应商每次都会派出一个团队与我们进行交流，销售、技术人员都会在这个团队里，可以明确地感知到销售和技术人员在表达能力上有着天壤之别：销售人员会告诉你他们什么都有，什么都可以做；而技术人员则要么不表达，一表达就陷入技术细节的怪圈，让旁人无法理解。一般的沟通过程由同行销售主导，优秀的销售人员会在客户与技术工程师之间搭建一条沟通之桥，而差劲的销售人员则会将沟通彻底搞砸。

某一商业组件曾经出现故障，供应商组织团队到现场处理问题，经过一轮分析诊断后大家在谈判桌上进行了讨论，下面是其中的一则笑谈。

甲方：“请厂商尽快提供应急方案！”

销售：“我们已有完善的应急方案，但由于之前我们在架构上做了很大的妥协，贵公司又对成本进行了严格控制，因此无法保证应急方案可行。”

技术陷入了沉思，怎样设计一个完善的应急方案呢？

甲方：“为什么连续几天出现异常却一直无法找到原因？”

销售：“因为每次异常的具体情况都不一样，暂无法做到有针对性地错误排查。”

技术陷入了沉思，查了几天出现问题的原因却毫无线索，到底问题出在哪里呢？

甲方：“下次出现故障应该怎么办？”

销售：“因为我们的应用是跑在微软操作系统上，因此得同时找微软。”

技术陷入沉思，如何部署一个详细的异常捕获方案呢？

甲方：“你们是否升级给了后线？”

销售：“经过长时间诊断，基本可以断定与我们的产品有关系的概率非常低，所以没有上报后线。”

技术陷入沉思，升级给后线也没用，他们解决不了问题呀！

甲方：“你们可以倒闭了。”

销售目瞪口呆。

技术：“我，我，我们会找到根本原因的。”

在整个过程中技术工程师只说了一句话，而销售的错误表述将甲方彻底激怒，最终谈崩。

IT 技术是一个知识密集型的工作，这个行业的人员学习了大量的专业技能知识。这些知识还会随着时间的变化而不断更新，技术人员需紧随步伐，不断地吸收知识，相反的是他们缺乏展现的机会，这种能力渐渐被弱化，以至于关键时刻丧失了对问题的基本表述能力。IT 技术比其他工作岗位都更需要一个开放、自由的工作氛围，一个可以随时与人互动、自我展示的工作平台，IT 技术人员需要在这样的环境中增加展现的机会，提升自己的表达能力。

我们要将 IT 放在一个开放、自由的平台上，什么样的平台才是开放、自由的呢？企业社交平台是什么？

企业社交平台是建设在企业内部的，是由同一部门、阶层或者全部员工构成的内部工作平台和社交平台。员工之间通过自由的知识分享、工作互动来促进交流、增进彼此的了解、建立信任关系，从而提高工作效率。IT 运维人员应当从书本、机器中走出来，向前多迈一步，展现自己的能力，借助社交平台培养习惯。

安德鲁 P·麦卡菲是麻省理工学院斯隆管理学院数字商务中心的首席研究科学家，他于 2006 年首次提出了“企业 2.0”的概念。企业 2.0 即企业社交平台，是企业自发性社会化软件平台。社会化软件使“人机交互”变成“人人交互”。企业管理也在从“以数据为中心”向“以人为中心”转变。下面是这个平台下的一些技术特征。

- 搜索 (Search)：能够让用户找到自己想要的东西。
- 链接 (Links)：基于互联网的搜索引擎比大多数企业内部网的性能更好，因为链接是其主要因素。
- 写作 (Authoring)：沃德·坎宁安说过“我们每个人都有张扬的天性，我要让没有出版过（作品）的人发现，写作是一种乐事。”并不是要发现那些还没有成为“莎士比亚”的普通人，而是让他们将知识、视野、经历、一桩真相、一段剪影、一个连接……写下来；写作是引出这些奉献内容的过程。
- 标签 (Tag)：为了建立大众分类机制，网站使用标签才能保存用户访问平台的信息，让信息工作的过程和模式更加可见。
- 通知：类似于更新提醒如博客提醒，一般由博客标题和指向新文章的链接组成。有了 RSS，用户就不再需要不停地上网检查更新；只要检查聚合器，单击感兴趣的标题，就能进入新内容所在的页面浏览。

麦卡菲通过对 VistaPrint、Serena 软件、美国情报系统和谷歌四个团队工作的案例进行分析，证明企业社交平台具有以下价值。

- 群体编辑：设置不同类别的人员通过个体的劳动来写作或者完成某项任务，解决了

版本控制和高成本平台两大问题。

- 创作：以一种持久的便于查询的方式共享知识、专长和经验，解决新员工及时融入工作、老员工突然离职、交接不利等问题。
- 广播搜索：指在一个公开论坛提出问题并希望得到答复，在解决问题的同时也有利于员工找到合作者。
- 网络的形成和维护：指能够帮助员工建立起对自己有价值的社会网络，把潜在性的联系转换成实际存在。
- 群体智慧：指从分散的群体行为中得到答案，所谓“群众的眼睛是雪亮的”正是来自于此。
- 自我组织：指在实施期间员工能力的提高，在这个过程中并不需要上级的指导。

### 15.4.2 目标管理，做好绩效

要想获得好的绩效，就要首先弄明白组织需要什么，自身需要什么，将两者的目标有效结合，才能获得高绩效。自我技能提升与团队绩效提升是两回事，企业的首要目标是盈利，因此不要期望企业将个人发展放到第一位，只有自己才能对自己负责。

要明确自己的绩效责任与目标及组织对自己的绩效期望，在做之前要明白我们是做什么的，为什么要做，结果是什么，如何衡量结果。在一个运维体系中，你的考核指标很可能会围绕你处理的工单、变更的数量、质量，同时可能参与一些重要项目。

请记住，在共同目标上遇到困难时，你的直线领导是最支持你的人，寻求并获得他的帮助，争取所需资源，包括责权、费用、工具等，不要一个人埋头苦干不出声。

做好对定期工作绩效的度量：工单数量是否增加；项目进度是否正常；重要任务是否偏离目标，及时获取来自上级的评价、指导与认同，多与上级沟通交流，保证目标的准确性。

### 15.4.3 知识管理，人员成长

IT 运维人员在成长的过程中会不断积累知识，如何对知识进行有效管理直接影响个人职业道路的发展。知识管理的一个基本问题是对知识的分类，知识一般分为原理知识（Why）和技能知识（How），从认知角度出发，原理知识通过文件、形象或其他精确的沟通过程来传授，但技能知识的获得却只能依赖于自身的体验、直觉和洞察力。

在实际运用中，技能似乎比原理更加重要，甚至有技能的人更懂原理，较之无技能的人更容易成功。理由是技能为专业可操作性知识，在运维中强调动手能力，这种能力需要通过实践积累。而一般认为懂得原理的人更加聪明，原理是普遍知识，能够以不变应万

变，知其然亦知其所以然，知其所以然才能教授他人。对 IT 运维人员的知识管理而言，对原理进行总结归纳再掌握一门具体的技能才是硬道理。

一般将个人知识管理分为收集、组织、分析、分享四个步骤。

- 收集：首先要确定个人的信息需求及其来源，选择合适的信息检索技巧和方法。通过书籍、互联网搜索、动手实践来获取我们需要的数据，但其尚未转化为知识，之后必须能判断这种信息与自己所关注问题的相关程度，从可信度、准确度、合理性等方面进行确认。
- 组织：在过滤无用和相关度不大的信息资源后，有效地存储信息，将信息做好归类，建立信息之间的联系，方便以后查找和使用。
- 分析：分析信息就牵涉到如何对数据进行分析，以提问、回答的方式加深自己的理解，并从中得出自己的结论。
- 分享：将自己的归纳总结通过个人博客、培训、会议的方式分享给大家，在分享中得到进一步提升。

目前在市面上流行多种笔记工具，例如有道云笔记、印象笔记、微软 OneNote 等，可以快速收集、组织生活及工作上的各种图文资料，如今大部分产品拥有云端同步功能，可以让你在任何设备上都能互通使用所有资料。

思维导图主要帮助我们进行思维发散，它能够记录下我们头脑里一闪而过的概念，并将它们归类梳理。Freemind 是一款开源思维导图工具。

在组织内部需要一棵知识树，通过协同的方式将知识共享到其 Wiki。

Blog 是分析与分享的重要战地，如果还没有开始维护个人博客或者有书写博客的习惯，则请尽快建立一个。

整理好你所要教授的材料，找一间会议室，准备好投影，将知识分享给你的伙伴。

最后让我们看看知识管理的公式，如图 15-10 所示，知识管理是人吸收信息并通过分享得到升华。

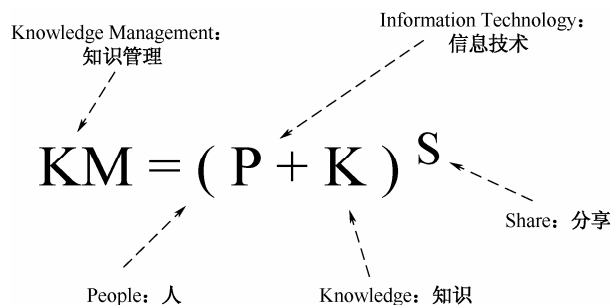


图 15-9 知识管理公式



## 1. Wiki 与 Markdown

只有将组织内的个人知识链接到一起，形成系统化的知识库才能产生知识最大效应。这个链接工作是一个漫长而耗时的过程，人们很快发现互联网的超文本标记语言（HTML）并不利于个人创作，对于习惯于命令行、文本窗口的系统管理员来说，纯文本的编辑方式才是他们创作利器，于是人们定义一种新的标记语言，它具备纯文本的简易的编辑方式，同时能快速地转换成 HTML，保存精美、可读的格式发布到互联网上。Wiki 和 Markdown 是这种新语言的代表。

基于 Wiki 语法的协同创作系统有着如下特征：

内容链接。在简单易用的语法下，人们通过文本编辑器创作知识页面，当需要关联内容时，只需要写成“[关联词条]”即可产生内部链接。对于内容相同，名称多样的词条内容可以进行内部归并，指向同一个页面。

版本控制。知识内容的变化受控，历史版本信息被记录下，可以快速地查阅与恢复历史版本，便于校对与回滚。

快速检索。系统本身是一个文档存储数据库，提供内容索引与检索功能，能够依据关键词查找词条内容。

如果要选择一个开源的 Wiki 系统作为内部知识管理的话，那必然是 MediaWiki，它被知名的维基百科（Wikipedia）所采用，风靡全世界。其由 PHP 编程语言写成，可使用 MySQL、MariaDB、PostgreSQL 或 SQLite 作为其关系数据库。



图 15-11 MediaWiki 知识管理

Markdown 是简明语法的另一分支，笔者更偏好于使用 Markdown，它在技术人员中的流行程度高于 Wiki，主流的博客平台、内容管理器都支持 Markdown 语法，例如 WordPress、Joomla、Drupal、Github 等。

笔者习惯使用 Markdown 进行个人知识的创作、整理，在 Windows、OS X 平台上都有相应好用的 Markdown 编辑器，支持实时浏览，如 CuteMarkEd、markdownpad、Mou 等。虽然 Wiki 与 Markdown 语法类似，但每次转换却是一个烦人的过程。对于同一件事，掌握

一种标准即足够（Python 的理念），笔者在需要将知识发布到 Wiki 上时，会使用 pandoc（<http://pandoc.org/>）自动地将 Markdown 转换成 Wiki。

## 2. 知识问答

除了通过 Wiki 方式汇集内部知识外，一个互动的知识问答平台在企业内部也是必须的，在这里大家提问，并邀请该领域专家进行回答，通过交流讨论产生新思路。

Question2Answer 是开源的知识问答系统，其也是用 PHP 语言写成，使用 mysql 数据库。在功能上包括：内容搜索、问题分类与打标、问题投票与评论、用户间信息订阅等，完全满足企业内部知识问答需求。目前超过 1 万个网站采用了 Question2Answer，它支持 40 多种语言。

# Question2Answer

图 15-12 Question2Answer 知识问答

## 15.4.4 时间管理，个人效率

### 1. 番茄工作法

行者问老和尚：“你在得道前做什么？”老和尚说：“砍柴、挑水、做饭。”行者问：“那得道后呢？”老和尚说：“砍柴、挑水、做饭。”行者又问：“那何谓得道？”老和尚回答说：“得道前，砍柴时惦记着挑水，挑水时惦记着做饭；得道后砍柴即砍柴，挑水即挑水，做饭即做饭。”

这则寓言反映了时间管理中的专注思想，在一个时间点上专注于一件事。运维人员常常会在做一件事时被人打断，需要同时响应几个事件请求，年末做总结时，却发现自己这一年似乎什么都没有做。

在这里推荐老外的“番茄工作法”，这是一种不错的时间管理办法，简单易行。它把工作时间按 25 分钟一次进行切片，这期间不允许被打断，否则此番茄时间作废，高效工作 25 分钟之后，计时器响铃，停止工作，休息 5 分钟。合计一个完整的番茄时间为 30 分钟，一轮番茄时间由 3~4 个番茄构成，之后进行 15~30 分钟的休息。一个番茄是一个被设定为不被干扰的时间段，即便有请求进入，也只是简单地做记录，后续回复，这样一方面可以静下心来集中做当前的事情；另一方面也可以静下心来自我反思，对工作进行安排。有很多番茄工作法的 App 可下载供我们使用。

### 2. 四象限法则

根据四象限法则，一个人的工作可以分为四种类型：第一种是重要也很紧急的工作；第二种是重要但不紧急的工作；第三种是紧急但不重要工作；第四种是不重要也不紧急的

工作。我们很多人往往把 50%~60%的时间放在了处理紧急但不重要的工作上，而重要但不紧急的工作却用的时间很少。高效能人士会把 65%~80%的时间安排在重要但不紧急的工作上，由于他们把大部分工作都提前统筹和规划好了，所以其余象限的工作自然而然就减少了。

第二象限的重要但不紧急的工作包括对问题的发掘与预防、持续学习、关于团队的建设、真正有效的授权、确定自己的个人使命、长期的职业生涯规划、个人使命等。如图 15-10 所示。

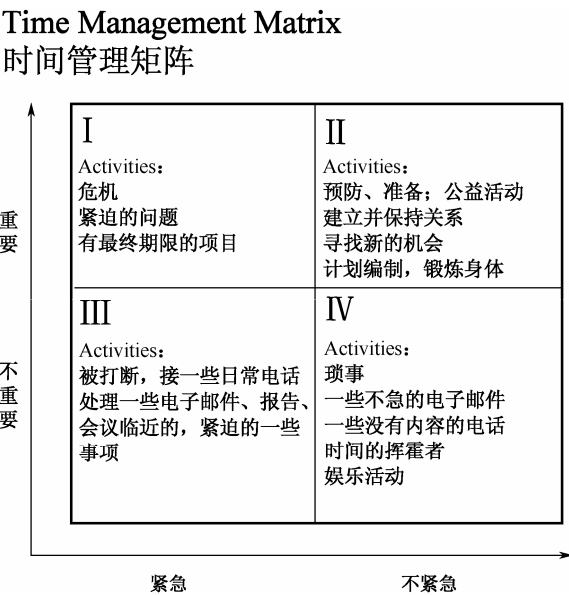


图 15-13 四象限法则

## 15.5 PaaS下的运维发展之路

PaaS 时代的来临，对运维职业发展将产生深远影响，一个严重的误区是认为云计算将彻底取代运维行业，为了回答这个问题，请让我们追溯历史，回顾运维职业的发展过程。

运维并不是一门历史悠久的职业，它是由系统开发人员发展而来，只有系统相关专业、熟悉底层资源、具备编程能力的才能胜任最早的运维，对他们的称呼一般是系统管理员。在 IT 发展之初，企业使用 IT 技术支撑内部办公，它并不会参与到核心业务中，运维人员负责内部电脑的安装、调试与连通。随着 IT 技术的突飞猛进，ERP 企业资源计划系统参与到了企业核心业务中，企业对 IT 的依赖不再局限于一般性质的办公需求，而是逐渐将核心业务流程、业务数据放到 IT 系统上，这时对运维人员的要求变高，除了掌握必要的专业知识外，在运维管理上要保证服务的可靠性。进入互联网时代后，IT 从之前的企业内部运

作参与者的角色直接转变成核心业务渠道，业务流量的入口都要依靠 IT，IT 直接面向终端用户。以 IT 技术为核心竞争力的企业在互联网时代实现了飞速增长，电子商务、游戏、即时通讯、搜索、视频等各个流量入口领域均被巨头占领，运维人员的价值又一次得到提升，IT 也从原来的成本中心一跃成为企业核心竞争部门。

我们可以看到在 IT 发展的过程中，对运维的要求也在不断提高。云计算、大数据、物联网以及移动互联等无一不是这个时代向前发展的标志，只要 IT 贴近用户，就会产生更多的数据、发现更多的需求，运维则愈加重要。

运维职业发展的三个硬道理是：

不变应万变。要做到以不变应万变，就必须掌握业内最基础、最稳固的知识点，打下扎实的基础。相对于开发应用框架、前端 UI 的变化，存储、计算、网络三大资源知识是非常稳固的，即便是变化也一定是建立在基础原理之上。互联网变化之快，新技术层出不穷，运维人员不能太过于跟风，一定要看清事物背后的本质，与基础原理相联系，深入底层内部思考，这样才能做到万变不离其宗。以 Linux 操作系统为例，运维人员并不需要将所有发行版的安装、命令等背诵如流，而是精通一到两种，并通过操作系统的运行原理来解释一切问题。

精通编程。不会编程的运维人员不是好运维，在开源风潮涌现的年代，可以预见未来对运维人员开发能力的要求会非常之高。系统开发与应用开发在完全不同的两个维度，系统开发更贴近于底层，掌握程序的运行原理对编程能力的提升有极大帮助，例如可执行文件的结构、在内存中的形态等。运维人必须精通一门编程语言，参与到社区，品读开源代码，养成编程习惯。引用 Linux 之父 Torvalds 的一句话“just for fun”，这是运维人看待编程应保持的心态。

敏锐观察力。时代依然在不断变化，运维人虽不必立即掌握每一项新出炉的技术，但他们必须保持对行业的关注度。预留一些时间给自己阅览社区新闻，积极参加线下社区活动，随着新技术的成熟以及自己的个人兴趣，在新兴领域投入必要的时间。

Larry Wall 是 Perl 语言的设计者，他属于运维的鼻祖，也就是系统管理员。当时 Larry 遇到了一个问题，如同我们现在遇到的一样。他需要在繁杂的内容中萃取文本信息，而手头的工具只有 awk、shell，这些工具可以帮助解决问题，但用起来却是那么痛苦，Larry 太懒了——如果用 awk 来做的话，要做大量工作，这让他无法忍受；Larry 也太急躁——awk 做起来很慢，他可等不及；最终他的高傲促使他完成了一个壮举，设计一门新语言——Perl，造福整个社区。是的，你会发现运维时代在变，但同样的故事还在发生，你是否已做好准备？